# Lifetime Safety: Preventing Leaks and Dangling

## I. Approach  II. Informal overview and rationale

Version 0.9.1.2

Herb Sutter and Neil MacIntosh, 2015-12-02

## Contents

**Acknowledgments**

Thanks especially to **Bjarne Stroustrup**. This work aims to help "finish the base model of C++" with its strong resource management model, and is codifying ideas that he has been developing for a decade and more. It also relies on features available uniquely, or most clearly, in C++ compared to other languages that do not rely on garbage collection. Stroustrup is a reliable source of both insight and pragmatism, and this paper is much clearer because of his feedback, including but not limited to the clear statement of principles in *Section I* and the Owners of Pointers examples.

Thanks also to Andrei Alexandrescu, Pavel Curtis, Gabriel Dos Reis, Joe Duffy, Daniel Frampton, Chris Hawblitzel, Steve Heller, Leif Kornstaedt, Aaron Lahman, Gor Nishanov, Andrew Pardoe, Jared Parsons, Arch Robison, Dave Sielaff, and Jim Springfield, for their comments and feedback on these ideas and/or drafts of this paper.

**Notes** This paper is a work in progress. It may well contain typos and minor inconsistencies as examples have been maintained by hand concurrently with the refinement of the model, although the examples are also run through the prototype compiler implementation. Bug reports are welcome – please open an issue.

There are three areas of known open work: (1) Iterate to refine the rules to address cases that generate excessive false positives when running the checker against production code; these will likely be addressed by adding specific rules, such as has already been done to address null tests (see II.5). (2) Finalize the shared ownership model. (3) Complete *Section III, Analysis rules* – the formal description to be used by implementations, which we anticipate producing as a separate companion paper in the coming months. (A small portion of Section III appears in this paper as Appendix 2, and is included here to make this paper better able to stand alone because some of the later examples use features, such as `move_owner`, that are from Section III.)

## Goal: Eliminate leaks and dangling for */&/iterators/views/ranges

We want freedom from leaks and dangling – not only for raw pointers and references, but all generalized Pointers such as iterators—while staying true to C++ and being adoptable:

1. **We cannot tolerate leaks (failure to free) or dangling (use-after-free).** For example, a safe `std::` library must prevent dangling uses such as `auto& bad = vec[0]; vec.push_back(); bad = 42;`.
2. **We need the raw efficiency of "just an address" pointers and references** especially when used locally on the stack as function arguments/parameters, return values, and local variables. For example, a safe `std::` library cannot just ban returning a reference from `vector<T>::operator[]`; it must still allow `vec[0] = 42;` to return and write through a raw reference. **We also need the flexibility of being able to write efficient user-defined iterators, ranges, views**, and other indirections, as pure libraries with comparable abstraction overheads to today's STL when used locally on the stack.
3. **We cannot tolerate run-time overheads**, such as widespread reliance on tracing garbage collection or other run-time instrumentation, as required in languages like Java, C#, D, Go and others where GC is mandatory or required to use the full standard library. That would defeat the purpose of passing "just an address" because the total cost to the program would include the cost of tracing collection. It also takes us out of the realm of C++, where "zero overhead" and "don't pay for what you don't use" are table stakes.
4. **We cannot tolerate significant source code impact**, such as heavy annotation as required in systems like Cyclone and Rust (see *Related work*). That would defeat both usability/comprehensibility and adoptability.
5. **We cannot tolerate excessive false positives**, or flagging as lifetime "errors" code that is not actually leaking or dangling. Our goal is that the false positive rate should be kept at under 10% on average over a large body of code; that is, 90% of diagnosed errors should be actual or latent errors.

This paper attempts to efficiently enforce leak-freedom and dangle-freedom statically at build time, including for existing C++ source code with little annotation and only simple reorganization. The key insight is that we can directly leverage C++'s existing strong notions of:

(a) scopes and lifetimes, including base and by-value member subobject lifetimes tied to the outer object;
(b) the compiler's awareness of private members, including full definitions of all types used by-value; and
(c) `const` to identify non-mutating accesses.

To distinguish owning from non-owning pointers, this design uses the GSL `owner<>` type alias to distinguish owning pointers and views, on which more complex owning types are built.

> **Note** Because `owner<>` is a type alias, it can be used to add build-time analysis information without disturbing any program types or ABI compatibility.

The design attempts to statically ensure leak-freedom and dangle-freedom for all indirection types including iterators and ranges, with no dynamic checking or reliance on GC, and therefore with identical space and time performance while continuing to use existing types including non-owning raw pointers and references.

> **Note** This paper focuses on lifetimes before and during `main`. As a future extension we may want to additionally consider lifetime errors during static destruction.

### Related work

Many thoughtful efforts to create a safe systems programming language don't meet the above requirements:

- *They handle only C, because "C is simpler."* We want a solution for C++; in fact, although the C *language* is simpler than C++, C++ *programs* are simpler to analyze than C because they contain much richer

information, notably through C++'s strong scope and ownership semantics. Several techniques in this paper exploit this information and can only be done in C++.

- *They incur run-time overheads.* Notably, many approaches rely on pervasive garbage collection, which incurs space (heap size) and time (collection pauses) overheads that do not fit C++'s zero-overhead "don't pay for what you don't use" design goal.
- *They rely on whole-program analysis.* This does not scale to real-world code sizes.
- *They require extensive annotation.* Excessive annotation is harder to write and usually difficult to reason about for the programmer. It also means that extensive necessary information is missing in the source code, which is less true for C++ – or that it is present but not used. Too much annotation blocks adoption because it is too invasive – it requires too much change before seeing benefits.
- *They invent a new language.* This could be tenable if it could be used side-by-side with C++ code with seamless interoperation, able to call C++ code directly without syntactic or code overhead, but no safe language we know of does this; some do provide good foreign function interfaces but only to C.
- *One or more of the above.*

Although this work is not derived from these efforts, we are aware of many of them. Below we note specific related work to compare/contrast approaches.

### Cyclone: Regions

Cyclone, an extension to C for memory safety, has the concept of growable regions, where a pointer can be associated with a region to ensure the pointer never outlives the region it points into. Regions can be implicit, but often need to be written explicitly in source code.

The result has advantages but is cumbersome to use. For an extension in this direction to be successful, the programmer should almost never be required to write these annotations, especially not on function parameters or on stack variables in calling code, and only rarely on library function signatures. Because C++ already has strong knowledge of scoped lifetimes for local and member variables, a key insight is that we can simply use the name of any scoped variable to denote its implicit lifetime 'region' without additional notation.

### Rust and System C# : Borrowing

Rust, and following it System C#, have the concept of borrowing a pointer or reference to an object and checking its lifetime statically. Both features are more general and complex than the one proposed in this paper.

System C# is built on the C# environment and object model, which carries capabilities and constraints not applicable to C++ and so naturally leads to a different design point. C# assumes that objects have lazy GC lifetime; we cannot, because that would violate our zero-overhead design constraint. C# objects are uncopyable and unscoped by default, and this constrains what can be expressed in the System C# model; instead, C++ already has strong knowledge of scoped lifetimes for local and member variables, and has statically known deterministic lifetime and destruction by construction and by default (e.g., default RAII vs. C# opt-in *using*).

Rust borrowing is part of a more general attempt to deeply enforce strong object ownership semantics pervasively, including with a number of built-in pointer types; instead, our design is a simpler feature focused on eliminating leaks and dangling only, while enabling similar pointer types (including the `std::` smart pointers with little or no modification) to still be built as libraries. Also, Rust chooses to treat the creation of a dangling pointer as an error; this seems too restrictive, especially because reusing pointer/iterator local variables is common and easy to make safe, so we feel that for local variables it is sufficient to diagnose only attempts to read (including dereference, copy from, and compare) dangling pointers.

# I. Approach and principles

The basic rules we teach the programmer are:

1. Prefer to allocate heap objects using (owning) `make_unique<T>`/`make_shared<T>` or containers.
2. Otherwise, use `owner<T*>` for source/layout compatibility with old code. Each non-null `owner<>` must be deleted exactly once, or moved.
3. Never dereference a null or invalid Pointer.
4. Never allow an invalid Pointer to escape a function.

The more detailed lifetime tracking system rules in the rest of this paper describe a means to soundly diagnose violations of rules 2, 3, and 4 at build time.

The approach in this paper is:

- **Local rules, statically enforced.** Doing all checks at build time locally within a function avoids run-time overhead while also keeping builds scalable. Whole-program guarantees are achieved when building the whole program, but without global analysis.
- **Identify Owners, track Pointers.** There are two kinds of indirection: Owners own what they point to and cannot dangle, and Pointers that do not and can dangle. The rules for Owners aim to ensure they clean up accurately and do not leak. The rules for Pointers aim to ensure they do not dangle.
- **Few annotations.** Deduction rules let us infer the vast majority of Owner and Pointer types: If a type X contains an Owner by value, then X is an Owner; otherwise if a type X is not an Owner and contains a Pointer by value, then X is a Pointer. Similarly for function calls, well-chosen default lifetime rules for Pointers passed as parameter and return values aim to avoid the need for explicit lifetime annotation for the vast majority of functions.

The principles behind the design are:

- **A Pointer may not outlive the object it points to.** The only exception is that local variables of Pointer type can be reused; they can dangle as long as they are not read (including dereferenced, copied from, or escaped) while dangling, which can be enforced locally.
- **We track the outermost object.** For an object held by value as a class member or array element, we track the enclosing object or array. For a heap object, we track its Owner.
- **When calling a function, a Pointer passed in as a parameter must be valid for the lifetime of the function.** This is enforced at the call site by disallowing passing a pointer the callee could invalidate.
- **When calling a function, by default Pointer parameters are independent.** This is enforced at the call site by disallowing passing a Pointer the callee could invalidate. Occasionally functions do something else; these require an annotation.
- **When calling a function, by default a returned Pointer is derived from the Owner and Pointer inputs.** This is enforced in the callee when separately compiling the called function's body. Occasionally functions do something else; these require an annotation.
- **Annotation is required to express a non-default lifetime or to say "trust me here."**

# II. Informal overview and rationale

This section is an informal tour of the design and design choices.

See *Section III, Analysis rules* (forthcoming this winter, separate document) for a full formal description.

## Indirections, Owners, and Pointers

An *Indirection* is an object that provides indirect access to another object. An *Owner* is an Indirection that owns the object it points to. A *Pointer* is an Indirection that does not own the object it points to.

We deduce these qualities for a type X as follows:

| X is a/an… | when X… | Examples | Can dangle |
|---|---|---|---|
| **SharedOwner**, shares ownership of the indirected object | contains a SharedOwner indirection member by value | `shared_ptr`, `raw_shared_owner<non-owner>` | no |
| **UniqueOwner**, owns the indirected object | is not a SharedOwner and contains a UniqueOwner member by value | containers, `unique_ptr`, `owner<non-owner>` | no |
| **Pointer**, doesn't own the indirected object | is not an Owner and contains a Pointer by value | raw * and &, iterators, ranges, views | yes |
| **not an Indirection** | is none of the above | `struct point { int x; int y; };` | no |

> **Note** `raw_shared_owner<>` is a type alias used as a building block for shared owners. The shared ownership model is currently being refined; more detail about shared ownership will be covered in future drafts of this paper, and in the upcoming *Section III, Analysis rules* formal description.

For example:

```
template<class T /*...*/>
class unique_ptr {                  // unique_ptr is a UniqueOwner...
    owner<T*> p;                    // ...because it contains one

    // ...

};

template<class T /*...*/>
class shared_ptr {                  // shared_ptr is a SharedOwner...
    raw_shared_owner<T*> p;         // ...because it contains one

    // ...

};

template<class T>
class container {                   // container is a UniqueOwner...
    unique_ptr<T[]> root;           // ...because it contains one

public:
    class iterator {                // iterator is a Pointer...
        container* cont;            // ...because it contains one
        // ...
    };

    // ...

};
```

```
class x {                   // x is a non-indirection...
    int i;                  // ...because it contains no Indirection
};
```

**Note** It is essential to distinguish Owners from Pointers, but this distinction needs to be made explicit primarily just for raw * and &; we believe we can infer the rest.

## Lifetime tracking for Pointers

A Pointer can be made lifetime-safe (e.g., no dangling use) by statically tracking what it points to, notably:

- the object it currently refers to; or
- the owner keeping the referred-to object alive.

For example, given

```
auto up = make_unique<int>(42);  // Owner, always valid
int* p = up.get();               // Pointer, can be invalidated
*p = 42;                         // ok
```

we want to capture that the pointer p is valid for the lifetime of the integer it refers to, which here is until up is destroyed or rebound; the latter happens when a non-const operation is performed on up:

```
up = something_else;             // A: invalidates p
*p = 42;                         // ERROR, p referred to an object owned by 'up'
                                 // before 'up' was modified on line A
```

A Pointer p that is a local variable may refer to something that it could outlive; this lets the programmer easily reuse p later in the local function. Although p could potentially dangle through invalidation, any actual invalidation is statically diagnosed, and once p is invalidated it must be destroyed or reassigned before any other use.

In the following examples (all examples showing raw * apply equally to &):

- 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).

## Points-to set (lset)

For a Pointer or SharedOwner p, let lset(p) denote what p refers to.

Let *lset* ("points-to set") be a set where each element is one of the following:

| | **Meaning** | **p is invalidated if** |
|---|---|---|
| **obj** | p currently refers to obj | obj is destroyed |
| **obj'** | p currently refers to an object owned directly by owner obj | obj is destroyed or modified by non-const use |
| **obj''** | p currently refers to an object kept alive indirectly (transitively) via owner obj | obj' is destroyed or modified by non-const use |
| **null** | p is invalid for any use until tested to be not equal to the null pointer constant | used without removing null from the list via a not-null branch |

| | | |
|---|---|---|
| **static** | p currently refers to a static object, or an object owned directly by a `const` static owner object | valid (until the end of `main`; may inject additional checking after `main` ends) |
| **invalid** | p is already invalid | always, it's already invalid |

> **Note** There is no provision for inventing names of "regions." All lifetimes are tied to existing objects that already have names and lifetimes, or in the case of shared references automatically synthesized from them.

`p` is valid to dereference as long as `lset(p)` does not contain `invalid` or `null`.

Notes:

- The set entries are interpreted as "or'd." For example, `lset(p) == {a,null}` denotes that `p` either refers to the object `a` or is null.
- Any other entries are redundant with `invalid`, so `(`*`anything`*`,invalid) == invalid`.
- More than two `'` means the same as two `'`. For example, `{a'''} == {a''}`.

Let `KILL(o)` mean to invalidate all occurrences of `o`, `o'`, and `o''` in existing lsets. For example, given `lset(p1) = x`, `lset(p2) = x'`, and `lset(p3) = x''`:

- `KILL(x)` invalidates all of `p1`, `p2`, and `p3`.
- `KILL(x')` invalidates `p2` and `p3`.
- `KILL(x'')` invalidates `p3`.

## 1. Aliasing: Taking addresses and dereferencing

Taking the address of an lvalue `x`, or of a data member or array element inside `x`, results in a (non-owning) raw pointer whose lset is `{x}`. A lset entry `o'` that refers to a data member `x.o` can be converted to a lset entry `{x'}`.

> **Note** A pointer to a local, or to a member, etc. can never be an owner. The only way to obtain an `owner<>` is from `new`. For any variable or other lvalue `x`, `&x` is still a `T*` (not `owner<T*>`), binding a reference to `x` is still a `T&` (not an `owner<T&>`) and you can't convert a `T*` to an `owner<T*>` implicitly without forcibly suppressing the lifetime rules.

### Example 1.1: Address of local variable and invalidation

For example:

```
int* p = nullptr;          // lset(p) = {null}
{
    int i = 0;             // lset(i) = {i}
    p = &i;                // lset(temp) = {i} → lset(p) = {i}
    auto i2 = *p;          // ok
    *p = 42;               // ok
}                          // A: KILL(i) → lset(p) = {invalid}
*p = 1;                    // ERROR, p was invalidated when i went out of scope
                           // at line A. Solution: increase i's lifetime, or
                           // reduce p's lifetime
```

> **Note** The solution in this and all examples is to change the scope of a local variable: to make the scope of a local pointer smaller (e.g., introduce additional pointer locals to separate flow) or to make the scope of a local variable bigger (e.g., move the destroyed local further out so the pointer isn't invalidated, or defer the mutation of a local owner).

### Example 1.2: Address of member variable or array element

Consider members and array elements:

```
struct mystruct { int m; } s;
auto p = &s.m;                  // lset(p) = {s}

int a[100];
auto p = &a[0];                 // lset(p) = {a}
```

### Example 1.3: lset(member variable or array element) = lset(enclosing object/array)

Consider members that are owners:

```
struct mystruct {
    int m;
    void f() {
        int* p = &m;            // lset(p) = {*this} (we are inside mystruct)
    }
} s;
int* p = &s.m;                  // lset(p) = {s} (we are outside mystruct)

int* a[100];
int* p = a[0];                  // lset(p) = {a}
```

### Example 1.4: Dereferencing

Consider dereferencing:

```
int   i  = 0;                   // non-indirection
int&  ri = i;                   // lset(ri) = {i}
int*  pi = &i;                  // lset(pi) = {i}

auto  s  = make_shared<int>(0); // Owner
auto* ps = &s;                  // lset(ps) = {s}

                                // Pointer
int** ppi = &pi;                // lset(ppi) = {pi}
```

Naturally therefore, dereferencing a pointer to pointer results pointer whose lset is substituted by the current lset of each entry – we are simply copying a pointer, including its lset. For example:

```
                                // IN:  lset(ppi)=={pi}, lset(pi)=={i}
int* pi2 = *ppi;                // lset(*ppi) == lset(pi) == {i}
                                // OUT: lset(pi2) = {i}

int   j = 0;
pi = &j;                        // lset(pi) = {j} – makes **ppi point to j,
                                //                  but only updates lset(pi)

                                // IN:  lset(ppi)=={pi}, lset(pi)=={j}
```

```
pi2 = *ppi;                         // lset(*ppi) == lset(pi) == {j}
                                    // OUT: lset(pi2) = {j}
```

## 2. Invalidation by modifying Owners

Modifying an Owner o invalidates anything whose lset depends on o'.

Dereferencing a Pointer no that could modify the target object invalidates anything whose lset depends on lset(no)' which means to add ' to each owner in the list. For example, if lset(x) = {a',b''}, then lset(x)' = {a'',b''}.

### Example 2.1: Invalidation by modifying Owners

For example:

```
auto s = make_shared<int>(0);
int* pi3 = s.get();                 // lset(pi3) = {s'} [more on this later]
s = make_shared<int>(1);            // A: KILL(s') → lset(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; // lset(sv2) = {sv}
vector<int>* vec = &*sv;            // lset(vec) = {sv'}
int* ptr = &(*sv)[5];               // lset(ptr) = {sv''}
*ptr = 1;                           // ok
                                    // track lset of:    sv2     vec     ptr
                                    //                   -----   -----   -----
                                    //             IN:   sv      sv'     sv''
vec->                               // same as "(*vec)." → *vec is sv'
     push_back(1);                  // non-const operation on sv' ⇒ KILL(sv'')
                                    //            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).                             // *sv2 is sv
       reset();                     // non-const operation on sv ⇒ KILL(sv')
                                    //            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).

### Example 2.2: Container of containers

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

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

*ptr = 0;                        // ok

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

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

## 3. Branches

When a Pointer is assigned to within a branch of an `if`, then at the end of the `if`'s scope we concatenate the lists at the end of each branch to record the "or'd" list of potential owners.

Similarly for `switch`, when a Pointer is assigned to within a path through a `switch`, then at the end of the `switch`'s scope we concatenate the lists at the end of each `break` path to record the "or'd" list of potential owners.

### Example 3.1: Invalidation in both branches

Both branches could invalidate. For example:

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

if(cond) {
    int i = 0;
    p = &i;                // lset(p) = {i}
    *p = 42;               // ok
}                          // A: KILL(i) → lset(p) = {invalid}
else {
    int j = 1;
    p = &j;                // lset(p) = j
    *p = 42;               // ok
}                          // B: KILL(j) → lset(p) = {invalid}
// merge → lset(p) = {invalid}

*p;                        // 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
```

### Example 3.2: Invalidation in one branch

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

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

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

A dereferenced pointer must be valid on all non-data-dependent control flow paths in the function leading to the dereference.

> **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).

### Example 3.3: Invalidation in neither branch

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

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

{
    int j = 1;

    if(cond) {
        p = &i;             // lset(p) = {i}
        *p = 42;            // ok
    }                       // no invalidation
    else {
        p = &j;             // lset(p) = {j}
        *p = 42;            // ok
```

```
    }                       // no invalidation
    // merge → lset(p) = {i,j}
    *p = 42;                // ok
}
```

## 4. Loops

A loop is treated much like an `if`, because as with an `if` there are only two paths to analyze: taken (the loop was entered at least once), and not taken (the loop was not entered). We do not do flow-sensitive analysis. However, processing a loop can require a second pass:

- We take one pass through to determine any changes to lsets used in the loop (on any path, as usual). This determines the full set of lsets affected on exit from any loop iteration.
- (Optional) If the exit set of lsets used in the loop is different from the entry set , we do one additional pass through the loop source starting with the new set of lsets to ensure that a subsequent loop iteration cannot rely on an invalidated lset modified during a previous loop iteration.

Note that this algorithm remains linear – we take at most two passes through the loop body.

### Example 4.1: Loops that do not change lsets

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

```
p = &a[0];                  // lset(p) = {a}
for( /*...whatever...*/ ) {
    // ...
    if( /*...whatever...*/ ) {
        // ...
        p = &a[i];          // lset(p) = {a}
        // ...
    }
    // ...
}
merge: lset(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.

### Example 4.2: Loops that do change lsets

If instead `p` could be pointed to another object during the loop, we would take the exit lset and then parse 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];                  // lset(p) = {a}
for( /*...whatever...*/ ) {
    // ...
    if( /*...whatever...*/ ) {
        // ...
        p = &b[i];          // lset(p) = {b}
```

```
        // ...
    }
    // merge → lset(p) = {a,b}
    // ...
}
merge: lset(p) = {a} /*before loop*/ ∪ {b} /*in loop body*) = {a,b}

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

### Example 4.3: Loops that invalidate

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

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

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

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

```
    }
    // merge → lset(p) = {null,b}
    // ...
}
// lset(p) = {null,a,b} still
```

### Example 4.4: Loops that allocate

Some loop bodies allocate:

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

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

if(p) *p = 42;              // ERROR, invalidated by assignment to p on line A
                            // (note conservative rule, because we don't accept
                            // owning raw * unless annotated owner<>

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];                  // lset(p) = {a}
unique_ptr<A> up;           // initially null
for( /*...whatever...*/ ) {
    // ...
    if( /*...whatever...*/ ) {
        // ...
        p = (up = new A).get();  // ok, lset(p) = {up'}
        // ...
    }
    // merge: lset(p) = {a, up'}
    // ...
}
merge: lset(p) = {a} /*before loop*/ ∪ {a,up'} /*loop body*/ = {a,up'}

// that's different from entry, so parse loop one more time with {a,up'}:
for( /*...whatever...*/ ) {
```

```
    // ...
    if( /*...whatever...*/ ) {
        // ...
        p = (up = new A).get();  // ok, lset(p) = {up'}
        // ...
    }
    // merge: lset(p) = {a,up'}
    // ...
}
// lset(p) = {a,up'} still
if(p) *p = 42;              // ok
```

## 5. `null`

When a branch can be entered only on success of an explicit test for `p` being not the null pointer constant (regardless of the complexity of the conditional expression), we remove the `null` dependency in that branch.

Determining whether a particular conditional subexpression is required to enter a branch is done "as if" by the conditional expression were rewritten as follows, applied recursively until fully simplified:

- A branch of the form `if(a && b){…}` is treated as `if(a){ if(b){…}}`.
- A branch of the form `if(a || b){…}` is treated as `if(a){ if(b){} else{…}}`.
- A conditional expression involving a `constexpr` function does not evaluate the `constexpr` function unless that evaluation is required by the language (i.e., appears in a `constexpr` context).
- A conditional expression of the form `arr[i]` for an array `arr` is treated as testing `lset(arr)`.

### Example 5.1: Removing `null` from a lset to dereference successfully

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

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

### Example 5.2: Replacing `null` in a lset with a valid object

For example, if a Pointer might be null, code can test for null and replace it with non-null. Here "…" means any other set contents:

```
int i = 0;
int j = 0;
```

```
int* p = rand()%2 ? &j : null;   // lset(p) = {null, j}

if(!p) {                         // in this branch, lset(p) = {null}
    p = &i;                      // lset(p) = {i}
}

// in implicit "else /*(p not null)*/ { /* no change to p */ }", lset(p) = {j}

// merge lset(p) = {i} ∪ {j}

p->foo();                    // ok, lset(p) == {i,j} does not contain null
```

Note that there is an implicit `else /*(p not null)*/ { /* no change to p */ }` branch, which is handled as in Example 5.1 – that is, in the `else` we have removed `null` as an option, so `lset(p) = {j}` there. At the end of the branches, we merge the results – `lset(p) = {i}` ∪ `{j}` from the true and false branches respectively, and end up with `lset(p) == {i,j}`. So code can naturally test for null and replace it with a valid entry.

## 6. `throw` and `catch`

### Example 6.1: `catch`

A `try` block is treated much like any other block, but a `catch` block is treated specially. Without statically knowing where the exception was raised, we treat the `catch` block as if it could have been entered from *every* point in the `try` block where an exception could have been raised. Thus we record all potential invalidations in the `try` block (as any of them may have executed) and remove any revalidations in the `try` block (as potentially none of them have executed.)

> **Note** Asynchronous exceptions are orthogonal to this question.

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

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

try {
    int j = 0;
    p1 = &j;          // A: lset(p1) = {j}
    f();
    p1 = &i;          // lset(p1) = {i}
    g();
    p2 = &j;          // B: lset(p2) = {j}
}                     // KILL(j) → lset(p2) = {invalid} in normal control flow

catch(...) {          // merge try's invalidations, ignore try's revalidations
    *p1 = 42;         // ERROR, invalidated by assignment to p1 on line A
    *p2 = 42;         // ERROR, invalidated by assignment to p2 on line B
}
```

### Example 6.2: `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 `static`. For example:

```
static gi = 0;

void f() {
    int i = 0;

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

## 7. Calling functions: Arguments and in/inout parameters

By default, objects and indirections passed to a function are assumed to be independent. This means that:

- In the function body, by default a Pointer parameter `param` is assumed to be valid for the duration of the function call and not depend on any other parameter, so at the start of the function `lset(param) = param` (its own lifetime) only.
- At a call site, by default passing a Pointer to a function requires that the argument's lset not include anything that could be invalidated by the function.

### Example 7.1: Passing indirections

For example:

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

// At call sites
//
int gi = 0;
shared_ptr<int> gsp = make_shared<int>();
int main() {
    // passing global and local objects
    f(&gi);          // ok, lset(arg) == {gi}, and gi outlives the call
    int i = 0;
    f(&i);           // ok, lset(arg) == {i}, and i outlives the call
    f(gsp.get());    // ERROR, lset(arg) == {gsp'}, and gsp is mutatable by f
```

```
    auto sp = gsp;
    f(sp.get());     // ok, lset(arg) == {sp'}, and sp is not mutatable by f
    g(sp, sp.get()); // ERROR, lset(arg2) == {sp'}, and sp is mutatable by f
    g(gsp, sp.get()); // ok, lset(arg2) == {sp'}, and sp is not mutatable by f
}
```

**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.)

### Example 7.2: 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 `[[lifetime(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 `[[lifetime(iter1)]]`.

Result:

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

    template <class InputIterator>
    void insert(InputIterator first,
                InputIterator last [[lifetime(first)]]);

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

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

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

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

m.insert(m.begin(), m.end()));         // ERROR, lsets=={m'}, and m is mutatable
                                       // by m.insert [per earlier rule]

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

m.insert(m2.begin(), m2.end());        // ok, lset == {m2'}, and m2 is not
                                       // mutatable by m.insert
```

**Note** This statically diagnoses several common classes of STL iterator bugs.

## 8. Calling functions: Return values and out/inout parameters

The goal of these defaults is to minimize total annotation, and to be sound when both the caller and callee are compiled separately under the `lifetime` profile. Any default lifetime that is incorrect will be diagnosed when compiling the callee body.

At the call site, when calling a function that produces (return value or out/inout parameter) a Pointer called `ret`, by default `lset(ret)` is derived from the function's Owner and Pointer arguments as in the following table.

If the only Owner arguments are passed by `&&` or non-`const&`, then treat all `owner const&` parameters as-if they were `non-const&`.

For each argument `arg` that matches one of the cases below, concatenate one entry into `lset(ret)` as shown; if after thus processing the arguments `lset(ret)` is empty, then `lset(ret) = {static}`.

Notes:

- `owner const&` arguments are intentionally excluded by default. They are asking for rvalues.
- `owner&&` arguments are intentionally excluded always. They are begging for rvalues.
- "Indeterminate" means that `arg` is a `this` pointer in an overridable `virtual` function (neither the function nor the class is `final`) in a class that is not a SharedOwner. This is considered indeterminate because a further-derived class could change the indirection category of the type.
- `lset(arg)'` means to add one `'` to every Owner in the set. For example, `{a,b',c'',null}' == {a',b'',c'',null}`.

| **`arg` is a** | **Passed by** | **`lset(ret)` ∪=** | **Examples** |
|---|---|---|---|
| SharedOwner of UniqueOwner | value, `&` or `&&`<br>`*` | `lset(arg)'`<br>`lset(*arg)'` | `int* f(shared_ptr<vector<int>>&);`<br>`int* f(shared_ptr<vector<int>>*);` |
| SharedOwner of other | value, `&` or `&&`<br>`*` | `lset(arg)`<br>`lset(*arg)` | `int* f(shared_ptr<X>, shared_ptr<Y>&);`<br>`int* f(gc_ptr<int>*);` |
| UniqueOwner of UniqueOwner | value, `const&`, `&&`<br>non-const `&`<br>`*` | `{}`<br>`arg''`<br>`(*arg)''` | `int* f(vector<vector<int>>);`<br>`int* f(vector<vector<int>>&);`<br>`int* f(set<int>*);` |
| UniqueOwner of other | value, `const&`, `&&`<br>non-const `&`<br>`*` | `{}`<br>`arg'`<br>`(*arg)'` | `int* f(vector<int>, const string&);`<br>`int* f(unique_ptr<int>&, string&);`<br>`int* f(set<int>*);` |
| other Pointer | value, `&` or `&&`<br>`*` | `lset(arg)`<br>`lset(*arg)` | `int* f(int*, int*&);`<br>`int* f(int**);` |
| indeterminate | `*` *(that is, `this`)* | `{}` | `struct base {`<br>`  virtual int* f();`<br>`};` |

### Example 8.1: Owners

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

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

### Example 8.2: `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& min(const T& a, const T& b) { // return lset = lset(a) ∪ lset(b)
    return a<b
           ? a       // ok, lset(a) is within lset(a)∪lset(b)
           : b;      // ok, lset(b) is within lset(a)∪lset(b)
}

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

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

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

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

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

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

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:

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

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

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

In safe code, just attempting to create the `bad` reference is a build-time error. The reference is unusable and cannot be rebound to make it usable; there is no reason to allow this.

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.

### Example 8.3: Explicitly overriding defaults

These defaults are useful and handle most cases, including `std::min`, `std::move`, `std::forward`, standard containers' member functions, standard algorithms, Meyers Singletons, and more. In cases where the function does something different by returning a Pointer to an object accessed on another path, such as heap objects accessed indirectly from static roots and new heap objects, explicitly write the lset using `[[lifetime(lset)]]`. (See examples later in the paper, notably 8.2 and 10.)

### Example 8.4: 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.

```
int* f() {
    int i = 0;
    return &i;       // lset(&i) = {i}, then KILL(i) → lset(ret) = {invalid}
                     // ERROR, cannot convert lset(ret)=invalid to ()
}

void g(int*& pi) {
    int i = 0;
    pi = &i;         // lset(pi) = {i}
}                    // KILL(i) → lset(pi) = {invalid}
                     // ERROR, pi is non-const& so lset(pi) must be {} on exit,
                     // and cannot convert {invalid} to {}
```

### Example 8.5: 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.

```
int* f(int* pi) {
    int i = 0;
    return cond : pi : &i; // lset(expr)={pi,i}, KILL(i) → lset(expr)={invalid}
                           // ERROR, cannot convert lset(ret)={invalid} to {}
}

void g(int*& pi, int* pi2) {
    int i = 0;
```

```
    pi = cond ? pi2 : &i;
}                           // KILL(i) → lset(pi) = {invalid}
                            // ERROR, pi is non-const& so lset(pi) must be {} on
                            // exit, and cannot convert invalid to {}
```

### Example 8.6: Calling a function that returns an indirection

```
int* f();                   // lset(ret) = {static}

int main() {
    int* p = f();           // lset(p) = {static}
    *p = 42;                // ok
}
```

### Example 8.7: Indirection returned from an owner member

Copying a non-owner indirection object from an owner member `mp` implicitly carries with it `lset = mp'`.

```
class smart_ptr_to_int {    // *this is an owner because ...
    owner<int*> p;          // ... it contains something known to be an owner

public:
    int* get() const {      // lset(ret) = {(*this)'}  (this is a * to owner)
        return p;           // lset(p) = {p'}
    }

    // ...
};

smart_ptr_to_int sp = /*...*/;
int* p = sp.get();          // lset(p) = {sp'}
*p = 42;                    // ok
sp = /*...*/;               // KILL(sp') → lset(p) = {invalid}
*p = 43;                    // ERROR, p was invalidated by assignment to sp on
                            // line A (non-const operation on sp)
```

> **Note** One operation that could invalidate the pointer and that can be called on a `const` pointer is `delete`. Therefore, in the `lifetime` profile, `delete` of a `const` pointer is not permitted except in the body of a destructor and then only if the pointer is a data member (and necessarily an `owner` data member per the other rules).

### Example 8.8: Owner of Pointer(s)

An owner of a non-owner has a lset, which applies to all the owned non-owners.

When the contained non-owner is assigned:

- If the owner `o` owns a single non-owner, such as with `unique_ptr<int*>`, the lset is replaced: `lset(o) = lifetime`.
- If the owner `o` potentially owns more than one non-owner, such as with `vector<int*>`, the lset is extended: `lset(o) = lset(o) ∪ lifetime`.

When the owner is assigned, the lset is replaced.

For example:

```
vector<int*> v;

void f() {
    int i;
    v.push_back(&i);            // ERROR, v outlives {i}
}
```

### Example 8.9: Local Owner of Pointers

Consider this example:

```
void compute1(vector<T>& v)
{
     span<T> buf = new T*[10];   // bad style, but not uncommon
     // fill buf with pointers into v
     delete buf;
}
// this is not a problem because buf doesn't escape
```

Correct. The only required change would be that `buf`'s type must be `owner<span<T>>`.

Elaborated example:

```
void compute1(vector<T>& v)
{
     owner<span<T>> buf = new T*[10];  // Owner of Pointers
     buf[0] = &v[0];             // ok, lset(buf) += lset(&v[0]) == v'
     delete[] buf.data();
}
```

### Example 8.10: Return Owner<> of Pointers

Consider this example:

```
T** compute1(vector<T>& v)
{
     T* buf = new T*[10];        // bad style, but not uncommon
     // fill buf with pointers into v
     return buf;
}
// this violates the "raw pointers do not own" rule, but doesn't
// actually leak pointers because buf's pointers are into v
```

Yes, both `buf` and the returned value must be `owner<span<T>>`.

Elaborated example:

```
owner<span<T>> compute1(vector<T>& v) // lset(ret) = {unref(v)'} by default
{
     owner<span<T*>> buf = new T*[10]; // Owner of Pointers
     buf[0] = &v[0];             // ok, lset(buf) += lset(&v[0]) == v'
     return buf;                 // ok, every entry in lset promotes
}
```

### Example 8.11: Return smart Owner of Pointers

Consider this example:

```
// Cleaner:
unique_ptr<T*> compute1(vector<T>& v)
{
     auto buf = make_unique<T*[10]>(); // or whatever
     // fill buf with pointers into v
     return buf;
}
```

Yes, this works as written. Elaborated example:

```
unique_ptr<T*> compute1(vector<T>& v) // lset(ret) = {v'} by default
{
     auto buf = make_unique<T*[10]>(); // Owner of Pointers
     buf[0] = &v[0];             // ok, lset(buf) += lset(&v[0]) == v'
     return buf;                 // ok, every entry in lset promotes
}
```

## 9. Transferring ownership

Analogously with assignment from Pointers: When an Owner `o1` is move-constructed or move-assigned to another Owner `o2` (of the same type), the ownership moves from `o1` to `o2`, and so in all lsets replace `o1` with `o2`.

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

{
owning<int*> o1 = new int(0);
pi = *o1;                       // lset(pi) = {o1'}
auto o2 = release_owner(o1);    // lset(pi) = {o2'} – note, no KILLs here
delete o2;                      // ok, must delete o2...
} // ... and this is also ok, need not (and may not) delete o1
```

## 10. Lifetime-`const`

In some cases, we will need to tag non-const member functions that are logically `const` for the purpose of lifetime invalidation.

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) [[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`, while 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) [[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.

# Appendix 1: Applied examples and experiments

## Additional examples

### Local and returned pointers

Consider this example:

```
int* p1;

int* f(int* p3, int i)
{
    int* p2 = &i;
    switch(i) {
    case 1:    return p1;
    case 2:    return p2;        // BAD
    case 3:    return p3;
    default:   return nullptr;   // a different problem
}

int* g()
{
    int x;
    auto p1 = f(&x,1);           // global
    auto p2 = f(&x,2);           // BAD
    auto p3 = f(&x,3);           // local
}
```

Correct. Here's how the example is processed:

```
int* p1;                         // lset(p1) is required to be always {static}

int* f(int* p, int i)            // lset(ret) == lset(p)
{
    int* p2 = &i;                // ok, lset(p2) = {i}
    switch(i) {
    case 1:    return p1;        // ok, {static} → lset(p) (the default rules
                                 //     assume f got the pointer from p)
    case 2:    return p2;        // ERROR, lset(ret) ⊄ {i}
    case 3:    return p;         // ok, lset(p) (what the default rules assume)
    default:   return nullptr;   // a different problem
}
```

### Getting sneaky

Consider this example:

```
// This is getting sneaky

int* glob;

template<class T>
void steal(T x)
```

```
{
     glob = x();
}
void f()
{
     int i;
     steal([&]{ return &i; });
}
int main()
{
     f();
     *glob = 7;
}
```

That's sneaky all right.

Prelude: Recall how lambdas are generated. Given:

```
int i;
steal([&]{ return &i; });
```

The code with the generated lambda is essentially:

```
int i;
struct __lambda {
    int& __i;
    __lambda(int& i) :__i{i} {} // store reference
    auto operator(){ return &__i; }
};
steal(__lambda{i});
```

With that in hand, first consider `f`: The lambda just generates a class with an `int&` member; this makes the lambda a Pointer. As we said earlier for `array<T>::iterator`, this has two effects: First, it allows conversion from the *reference* parameter's lifetime to the *member* variable, and it explicitly lets us infer that the lambda constructed using this constructor is valid for the lifetime of `i`, so the iterator instance produced has validity `lset(*this) = lset(i)`.

```
void f()
{
     int i;
     steal([&]                    // ok, lset(lambda) = {i}
           { return &i; });       // ok, return &member, so lset(ret)={i}
}
```

However, `steal` contains an error, because the only legal assignment to `glob` would be something with a lset known to be `{static}`.

```
template<class T>
void steal(T x)
{
```

```
    glob = x();                 // ERROR, lset(x()) is not {static}
}
```

### Return collection of pointers

Consider this example:

```
vector<int*> find_all(vector<int>& v, int i);
    // return pointers to elemnts of v with the value i

int* pp;

int* f()
{
    vector<int> v = {1,2,3,4};
    auto r = find_all(v,3);     // this is fine
    pp = r[0];                  // this is not fine
    return r[0];                // this is not fine
}

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

int* f()
{
    auto r = find_all(vv,3);    // this is fine
    pp = r[0];                  // this is fine
    return r[0];                // this is fine
}
```

Yes, except for two corrections, both of which we can enforce:

- `find_all` should take the vector by reference.
- The second-last line is incorrect because `pp` could be invalidated by modifying `vv`. It would be legal if vv were `const`.

Here is an elaborated example:

```
vector<int*> find_all(vector<int> v&, int i) { // lset(ret) = {v'} by default
    vector<int*> ret;
    for (auto& e : v)
        if (e == i)
            ret.push_back(&e);  // lset(ret) += {v'}
    return ret;                 // ok, lsets match
}

int* pp;

int* f()                        // lset(ret) = {static} by default
{
    vector<int> v = {1,2,3,4};
    auto r = find_all(v,3);     // ok, lset(r) = {v'}
    pp = r[0];                  // ERROR, can't expand {v'} to {static}
    return r[0];                // ERROR, can't expand {v'} to {static}
}
```

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

int* f()                        // lset(ret) = {static} by default
{
    auto r = find_all(vv,3);    // ok, lset(r) = {vv'}
    pp = r[0];                  // ERROR, can't expand {vv'} to {static}
    return r[0];                // ERROR, can't expand {vv'} to {static}
}
```

But add `const` and the second part works:

```
const vector<int> cv = {1,2,3,4};

const int* f()                  // lset(ret) = {static} by default
{
    auto r = find_all(cv,3);    // ok, lset(r) = {cv'}
    pp = r[0];                  // ok, {cv'} + const cv can expand to {static}
    return r[0];                // ok, {cv'} + const cv can expand to {static}
}
```

## Implementing and using `std::unique_ptr` for single objects

The following is intended to be a completely `lifetime`-safe implementation of `unique_ptr`. For exposition, we omit machinery like the deleter which can also be expressed in a safe way.

> **Note** For convenience, this code uses the `move_owner` and `release_owner` helpers, but these are not required. The code could with equal validity perform the naked deleting/assignment/null-setting suboperations explicitly.

### Declaration and data

The class begins as usual, using `owner` to declare the ownership of the pointer.

```
template<class T>
class unique_ptr {
    owner<T*> p = nullptr;
```

### Dereferencing: `get`, `operator->`, and `operator*`

The dereferencing operations just work.

```
public:
    T* get() const {            // lset(ret) = {(*this)'} (by default)
        return p;               // lset(p) = {(*this)'} (because it's a member)
    }

    T* operator->() const {     // lset(ret) = {(*this)'}
        Expects(p != nullptr);
        return p;               // lset(p) = {(*this)'}
    }

    T& operator*() const {      // lset(ret) = {(*this)'}
        Expects(p != nullptr);
        return *p;              // lset(p) = {(*this)'}
    }
```

### Aside: Potential new implicit conversion `operator T*`

Note that this means that smart pointers can now safely offer an implicit conversion `operator T*()`. It is a well-known (and often lamented) problem that smart pointers like `unique_ptr<T>` must not offer implicit conversions to `T*`, which is otherwise desirable for usability and substitutability, for two reasons that no longer apply in safe code:

- First, the `lifetime` issue: In unsafe code it is too easy to get a pointer without realizing we must take care that it does not dangle. This is not a problem under this proposal in `lifetime`-safe code, because we now track lifetimes by default and can prevent use of a dangling pointer, and so such incorrect uses would fail to build with a clear error message.
- Second, the `bounds` arithmetic issue: In unsafe code it is too easy for unintended and incorrect code to accidentally work when `smartptr` converts implicitly to `T*`. For example, the code `smartptr + 42` could convert `smartptr` to `T*` and then invoke the built-in `+` that takes `T*` and `int`, which was unintended, logically wrong, and potentially seriously wrong because it incidentally produces a wild pointer which in a slightly more complex example could further be silently dereferenced (e.g., `*smartptr + 42` vs. `*(smartptr + 42)`). However, this is not a problem in `bounds`-safe code because pointer arithmetic is banned in the `bounds` profile, and so such incorrect uses would fail to build with a clear error message.

The only drawback to taking advantage of this is that a smart pointer that relies on this safety to provide an implicit conversion to `T*` will be unsafe if *called from* non-bounds-safe or non-lifetime-safe code. Therefore the operator must be marked as available only when both the `bounds` and `lifetime` profiles are in effect.

So we could consider inserting the new convenience function:

```
operator T*() const [[enable_if_profiles(lifetime && bounds)]] {
    return p;
}
```

**Notes** This would be a departure from profiles being strictly subsets, as this would be an extension available only under the listed profiles. The spelling of `enable_if_profiles` is bikesheddable.

However, note how elegantly this matches the concept mentioned earlier in the `owner<>` section that an `owner<T*>` should be thought of as being an unencapsulated `unique_ptr<T>`: The `lifetime` safety rules do permit an implicit conversion from an `owner<T*>` object named `o` to a non-owner `T*` with lset `o'`. That operation is the equivalent of the above.

### Special member functions: Construction, destruction, copying, and moving

Consider construction and destruction:

```
unique_ptr() = default;

~unique_ptr() { delete p; }
```

This works because `p` is an `owner<T*>`. Note that the `delete` is impossible to forget: Without it, `~unique_ptr` would fail to build with the error that the `owner<T*> p` was not deleted.

**Note** On the other hand, if the type of `p` were instead a plain `T*`, the `delete` would be an error because explicit `delete` is not permitted in `lifetime`-safe code.

Copying is disabled and uninteresting:

```
unique_ptr(unique_ptr&) = delete;
unique_ptr& operator=(unique_ptr&) = delete;
```

Let's finish the special member functions by considering the move operations, which manually manipulate lifetimes but do so in a safe and checkable way.

```
unique_ptr(unique_ptr&& other)
  : p{ release_owner(other.p) }
{ }
unique_ptr& operator=(unique_ptr&& other) {
    p = move_owner(p, other.p);
}
```

### `reset` and `release`

Next, consider `reset`, which adopts a previously-owning raw pointer. This is inherently a `lifetime`-unsafe "trust me" function for two reasons: first, because owning raw pointers are always `lifetime`-unsafe; and second, because allowing implicit adoption would enable adopting the same raw pointer by multiple `unique_ptr`s which is lifetime-unsafe (e.g., would lead to double `delete`). If we try to write the usual `lifetime`-unsafe code, we'll get an error – the compiler is telling us that this is an unsafe operation:

```
// BAD naïve implementation, build time errors
void reset(T* ptr = nullptr) {
    T* old = p;             // lset(old) = {p'}
    p = ptr;                // KILL(p) → lset(old) = {invalid}
                            // ERROR, p outlives ptr
    delete old;             // ERROR, old invalid, and delete of non-owner
}
```

We could just suppress the `lifetime` profile on these two lines, but the correct solution is just to annotate that the parameter is an `owner`, because we are going to take ownership: Not only will this will prevent `lifetime`-safe calling code to pass the same pointer to `reset` on two `unique_ptr`s, which is desirable because having two `unique_ptr`s adopt the same object would be wrong, but it makes the body just work (though for convenience we'll use `move_owner`):

```
// Corrected implementation
void reset(owner<T*> ptr = nullptr) {
    move_owner(p, ptr);
}
```

**Note** Again this is a parallel with `owner<T*>` which does not allow assignment from a (non-owning) `T*`.

The constructor from `T*` does the same:

```
unique_ptr(owner<T*> ptr)
  : p{ release_owner(ptr) }
{ }
```

Similar reasoning applies to `release`: We annotate the return type with `owner<>`, which does not change the type, and this not only correctly documents that ownership is being moved to the caller, but it removes the need to suppress `lifetime` safety rules.

```
owner<T*> release() {
    return release_owner(p);
}
```

### swap

Finally, consider `swap`.

```
void swap(unique_ptr& other) {
    std::swap(p, other.p);
}
```

### Example: From StackOverflow

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 :

```
const A& rA = *myFun();    // A: ERROR, rA is unusable, initialized with 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 `lset(`*`temp_ref`*`) = `*`temp_up'`, then `temp_up` is destroyed which implies `KILL`(*`temp_up`*) → `lset(`*`temp_ref`*`) = invalid`, and finally that is copied to initialize `lset(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, lset(rA) = {local}
use(rA);                       // ok, we know that local is keeping rA alive
```

## Implementing gc_ptr<T>

Let's imagine we want a new smart pointer type called `gc_ptr` that points into garbage-collected memory. The implementation is similar to `unique_ptr`, except that `gc_ptr` is copyable so it needs a `raw_shared_owner`. Also, it deliberately does not `delete` the `raw_shared_owner` in its own member functions, so we have to be able to express those semantics.

```
template<class T>
class gc_ptr {
    raw_shared_owner<T*> p = nullptr;

public:
    T* get() const {
        return p;
    }

    T* operator->() const {
        return p;
    }

    T& operator*() const {
        Expects(p != nullptr);
        return *p;
    }

    operator T*() const [[enable_if_profiles(lifetime && bounds)]] {
        return p;
    }

    gc_ptr() = default;
```

Unlike `unique_ptr`, `gc_ptr` does not express unique ownership and so is copyable, and copying naturally doesn't modify the source object. Because it is efficiently copyable, it doesn't need distinct move operations. However, it doesn't `delete` its `owner` member, so we have to `[[suppress(lifetime)]]` to say that's okay:

```
    ~gc_ptr() {
        p = nullptr;            // ok for raw_shared_owner<>
    }

    gc_ptr(const gc_ptr& other) {
        p = other.p;            // ok for raw_shared_owner<>
    }

    gc_ptr& operator=(gc_ptr& other) {
        p = other.p;            // ok for raw_shared_owner<>
    }
```

Note that `reset` and `release` do not make sense for `gc_ptr`, so we omit them. And `swap` is unchanged:

```
    void swap(gc_ptr& other) {
        swap(p,other.p);
    }
```

## Implementing `unique_ptr<T[]>` for arrays

The specialization `unique_ptr<T[]>` for arrays is similar, but let's do a small upgrade to use a `span` for `bounds` safety.

```
template<class T, /*...*/> class span; // more on this later

template<class T>
class unique_ptr<T[]> {
    owner<span<T>> av;

public:
    span<T> get() const {
        return av;
    }
```

**Note** As an additional improvement for `bounds` safety, I chose to change the return type of `get()` to `span<T>`. Returning a raw `T*` would not be that useful under the `bounds` profile where the pointer could be used only as a pointer to a single object.

If we want strict compatibility with the current standard `unique_ptr<T[]>` interface, we could alternatively provide

```
T* get() const {
    return av.data();
}
```

but adopting the return type change above adds better `bounds` safety; in `bounds`-safe code the returned `T*` would be usable only as a pointer to a single object.

Instead of `operator->` and `operator*`, the array version of `unique_ptr` provides `operator[]`:

```
template<class T>
T& unique_ptr<T[]>::operator[](size_t pos) const { // lset(ret) = {(*this)'}
    Expects(pos < av.size());
    return av[pos];
}
```

The expression `av[pos]` gives a `T&` to an element of the `span` whose owner is the same as the owner of the `span`, namely `*this`.

**Note** Because the returned `T&` is `lifetime`-safe, combined with `av` also providing `bounds`-safety, we have complete memory safety for the returned `T&`.

Aside: Similarly to `unique_ptr` for single objects, we can provide an implicit conversion to `span`, which is safe to use if the caller is `lifetime`-safe (note `bounds` does not need to be required this time).

```
operator span<T>() const [[enable_if_profiles(lifetime)]] {
    return av;
}
```

The next few functions are essentially unchanged:

```
unique_ptr() = default;
```

```
~unique_ptr() {
    delete[] av.data();
}

unique_ptr(unique_ptr&) = delete;

unique_ptr& operator=(unique_ptr&) = delete;

unique_ptr(unique_ptr&& other) {
  : av{ release_owner(other.av) }
{ }

unique_ptr& operator=(unique_ptr&& other) {
    delete[] av.data();
    av = release_owner(other.av);
}

void reset(owner<span<T>> view = nullptr) {
    delete[] av.data();
    av = release_owner(view);
}

owner<span<T>> release() {
    return release_owner(av);
}

void swap(unique_ptr& other) {
    std::swap(p, other.p);
}
```

## Implementing `std::array<T>::iterator`

Now let's try containers and iterators.

Our focus will be on the iterator, so consider just a minimal subset of `std::array`, and how to implement its `iterator` to be `lifetime`-safe. Note `array` can happily remain an aggregate.

```
template<class T, std::size_t N>
class array {
    T[N] data;

public:
    T& operator[](std::size_t pos) {
        Expects(pos < N);        // (for bounds safety profile)
        return data[pos];        // lset(ret) = {*this}
    }
```

Now consider writing a new indirection: `array<T,N>::iterator`.

```
    class iterator {
        array* a;
        int pos = 0;
```

Now we can infer something useful:

```
        iterator(array& arr) : a{&arr} { }
```

We know that `iterator` is a Pointer because it contains one. This has two effects: First, it allows conversion from the *reference* parameter's lifetime `lset(arr)` to the *member* variable which would otherwise be a build time error because the lifetimes are unrelated. Second, and more significantly, it explicitly lets us infer that the iterator constructed using this constructor is valid for the lifetime of `arr`, so the iterator instance produced has validity `lset(*this) = lset(arr)`.

By capturing that `*this` refers to `arr`, it also changes the default lifetime of a returned pointer:

```
    public:
        T& operator*() const {
            Expects(pos < N);
            return a->data[pos]; // lset(ret) = {*a}
        }

        // ... ++, --, etc.
    }

    iterator begin() noexcept {  // lset(ret) = {*this}
        return iterator(*this);
    }

    // ...

};
```

Now consider the calling code:

```
array<int,100> array1;
int* ptr = nullptr;             // lset(ptr) = {null}

{
    auto i = array1.begin();    // lset(i) = {array1}
    auto j = i;                 // lset(j) = {array1}

    {
        array<int,100> array2;
        i = array2.begin();     // repointed, so now lset(i) = {array2}
        *i = 42;                // ok
    } // A: KILL(array2) → lset(i) = {invalid}

    *i = 42;                    // ERROR, i was invalidated on line A
    i = array1.begin();         // lset(i) = {array1}
    *i = 42;                    // ok, now i is usable again

    *j = 42;                    // ok
    ptr = &*j;                  // ok, lset(ptr) = {array1}
    ++j;
}

*ptr = 42;                      // ok: ptr's validity not tied to j
```

## Implementing span

Now let's try a simplified `span`.

```
template<class T> class span {
    T* a;
    int size = 0;
public:
    template<std::size_t N>
    span(std::array<T,N>& arr) : a{&arr[0]}, size{N} { }
                                // lset(*this) = lset(arr)
```

Again, the non-owner member enables conversion from the reference parameter's lifetime to the member variable, and lets us infer that the view constructed using this constructor is valid for the lifetime of `&arr[0]` which is the same as `arr`, so the view instance produced has `lset(*this) = arr'`.

Here are a few more functions, and for exposition let's just call all the parameters `arr`:

```
    template<std::size_t N>
    span(T (arr&)[N]) : a{&arr[0]}, size{N} { }
                                // lset(*this) = lset(arr)

    template<std::size_t N>
    span(std::vector<T> arr) : a{&arr[0]}, size{arr.size()} { }
                                // lset(*this) = lset(arr')
```

Let's throw in the copy constructor (the compiler-generated one would do the same thing):

```
    span(const span& other) : a{other.a}, size{other.size} { }
                                // *this copies lset from other
```

By capturing that `*this` refers to `arr`, it also changes the default lifetime of a returned pointer:

```
    T& operator[](int pos) const {
        Expects(pos < size);
        return a[pos];          // lset(ret) = {owners-of-this-object}
    }
    // ... ++, --, etc.
};
```

Now consider the calling code, and note how we naturally distinguish between `array` directly owning its memory (valid for the lifetime of the array) vs. `vector` indirectly owning its memory (valid until the vector is destroyed or modified):

```
array<int,100> array1;
int* ptr = nullptr;             // lset(ptr) = {null}

{
    span<int> i = array1;  // lset(i) = {array1}
    span<int> j = i;       // lset(i) = {array1}
    {
        vector<int> array2(100);
        i = array2;             // repointed, so now lset(i) = {array2'}
        i[0] = 42;              // ok
        array2.push_back(1);    // A: KILL(array2') → lset(i) = {invalid}
```

```
        i[0] = 42;              // ERROR, i invalidated by push_back on line A
    }

    i = array1;                 // lset(i) = {array1}
    i[0] = 42;                  // ok, now i is usable again

    j[0] = 42;                  // ok
    ptr = &j[0];                // ok, lset(ptr) = {array1}
}

*ptr = 42;                      // ok: ptr's validity not tied to j
```

## Implementing `std::vector<T>::iterator`

Consider just a minimal subset of `std::vector`, and how to implement its `iterator` to be `lifetime`-safe. This adds two twists over the `std::array` case from earlier:

- `vector` adds a level of indirection because the data is on the heap rather than as a member variable, and so the lifetimes are shorter because the heap data can be replaced.
- This code exercises composability, because we're going to try to reuse another owner type. Two candidates are `unique_ptr<T[]>` for arrays, and `owner<span<T>>`. Let's do the former.

Starting off:

```
template<class T>
class vector {
    unique_ptr<T[]> data;

public:
    T& operator[](std::size_t pos) {   // lset(ret) = {(*this)'}
        return data.get()[pos];
    }
```

And now `vector<T>::iterator`.

```
    class iterator {
        vector* v;
        int pos = 0;

        iterator(vector& vec) : v{&vec} { }

    public:
        T& operator*() const {
            return v[pos];             // lset(ret) = {(*v)'}
        }

        // ... ++, --, etc.
    }

    iterator begin() noexcept {        // lset(ret) = {(*this)'}
        return iterator(*this);
    }
```

Now consider the calling code:

```
vector<int> array1(100);
```

```
int* ptr = nullptr;             // lset(ptr) = {null}

{
    auto i = array1.begin();    // lset(i) = {array1'}
    auto j = i;                 // lset(j) = {array1'}

    {
        vector<int> array2(100);
        i = array2.begin();     // repointed, so now lset(i) = {array2'}
        *i = 42;                // ok
        array2.push_back(1);    // A: KILL(array2') → lset(i) = {invalid}
        *i = 42;                // ERROR, i was invalidated by
                                // "array2.push_back" on line A
    }

    i = array1.begin();         // lset(i) = {array1'}
    *i = 42;                    // ok, now i is usable again

    *j = 42;                    // ok
    ptr = &*j;                  // ok, lset(ptr) = {array1'}
    ++j;
}

*ptr = 42;                      // ok: ptr's validity not tied to j
```

## Implementing a tree-based container

### Handing out references

Transitive `shared_ptr`s maintain ownership. Note that using raw local `*` variables eliminates needless reference counting overhead.

```
template<class T>
class tree {
    shared_ptr<Node> root;

    struct Node {
        T data;
        shared_ptr<Node> left, right;
        weak_ptr<Node> parent;
    };

public:
    shared_ptr<T> get_root() {  // as a simple example
        assert(root.get());     // ok, lset(root.get()) = {root'}
        return shared_ptr<T>(root, root->data);
                                // lset(ret) = {root'}
    }
```

Note that this is safe because of the lifetime `root'` on the second argument.

```
    shared_ptr<T> get_leftmost_slow() {     // another simple example
        auto p = root;
```

```
        assert(p);
        while (p->left) p = p->left;
        return shared_ptr<T>(p, p->data);
    }

    shared_ptr<T> get_leftmost_fast() {      // another simple example
        Node* p = root.get();    // lset(p) = {root'}, not copying root
        assert(p);
        while (p->left)
            p = p->left.get();   // lset(p) = {root''}
        return shared_ptr<T>(p, p->data);
    }
};
```

Note that the last line in each function does this, without annotation:

```
        return shared_ptr<T>(p, p->data);
```

How can this constructor call be `lifetime`-safe? This is the `shared_ptr(r,ptr)` aliasing constructor, where "to avoid the possibility of a dangling pointer, the user of this constructor must ensure that `p` remains valid at least until the ownership group of `r` is destroyed" [ISO C++], and "it is the responsibility of the programmer to make sure that `ptr` remains valid as long as this `shared_ptr` exists, such as in the typical use cases where `ptr` is a member of the object managed by `r` or is an alias (e.g., downcast) of `r.get()`" [cppreference.com]. By now, all of that sounds familiar… we can spell it with a lifetime constraint:

```
template<class T, class Y>
shared_ptr<T>::shared_ptr( const shared_ptr<Y>& r, T *ptr [[lifetime(r')]] );
```

### Not handing out references

If we don't hand out references, `unique_ptr` suffices. Because we can't use `weak_ptr` here for the parent pointer, when we assign the parent pointer we need to say "trust me" via `[[suppress(lifetime)]]`.

```
template<class T>
class tree {
    unique_ptr<Node> root;

    struct Node {
        T data;
        unique_ptr<Node> left, right;
        Node* parent;

        Node(Node& parent_) { [[suppress(lifetime)]] parent = &up; }
    };

public:
    T get_root() {                    // as a simple example – now by value
        assert(root.get());
        return root->data;
    }

    T get_leftmost() {                // another simple example
        auto p = root.get();          // lset(p) = {root'}
        assert(p);
```

```
        while (p->left)
            p = p->left.get();          // lset(p) = {root''}
        return p->data;
    }
};
```

# Appendix 2: owner<> and raw_shared_owner<>

## owner<> and raw_shared_owner<>

### array_ptr alias

For compatibility and migration of older code that uses T* and cannot convert to use a smart pointer, we provide owner<T*> for a single object, and to distinguish pointer to arrays we provide the additional alias array_ptr to be used as owner<array_ptr<T>>.

Let

```
template<class T, int N> using array_ptr = T*;
```

be an alias that designates the pointer points to an array of length N, and allow conversions:

- from array_ptr<T,N> to span<T,N>;
- from owner<array_ptr<T,N>> to owner<span<T,N>>; and
- from raw_shared_owner<array_ptr<T,N>> to raw_shared_owner<span<T,N>>.

### Common rules

In the lifetime profile:

- new X returns an rvalue of type owner<X*>.
- new X[N] returns an rvalue of type owner<array_ptr<X,N>>.
- delete/delete[] cannot be called on any type except as permitted below.
- delete can be called on any owner<T> or raw_shared_owner<T>.
- delete[] can be called on owner<A> or raw_shared_owner<A> where A is an array_ptr or span.

**Notes** owner<span<T,N>> is allowed as well and is preferred. However, array_ptr is more compatible by being just an alias so that existing code using T* to point to an owned array can switch to owner<array_ptr<T,N>>.

As an extension, we could say that when T is an array_ptr, delete means to invoke delete[]. However, this is a semantic change that would require compiler implementation, whereas otherwise these rules can be implemented in any source build step (not necessarily in a compiler).

As a future extension to help C-style code, we could also consider providing some malloc compatibility along the following lines:

m_owner<X*> p = malloc(N) is legal iff sizeof(X)<=N and X is trivially constructible.

m_owner<array_ptr<X,M>> p = malloc(N) is legal iff sizeof(X)*M<=N and X is trivially constructible.

`m_owner<span<X,M>> p = malloc(N)` is legal iff `sizeof(X)*M<=N` and `X` is trivially constructible.

`free` is available for all of these with the lifetime state effect of `delete`.

Every `owner<T>` and `raw_shared_owner<T>` is always in one of the states {`valid`, `invalid`, `null`} and obeys the following state transition tables. In those tables:

- DECLARE(`x`) means `x` is declared. Its initial value must be set as a separate step.
- END_LIFE(`x`) means that `x` is destroyed (e.g., a local variable at the end of its scope).
- ESCAPE(`x`) means that `x` leaves the local function scope (e.g., is a modifiable `&` parameter and we are leaving the function body via `return` or `throw`).

### Unique ownership: `owner<>`

`owner<>` is a type alias that allows legacy and lowest-level data structures using owning raw pointers with unique ownership to be correctly identified as owners in an ABI-compatible way by not disturbing their type. `owner` is intended to be principally used for single heap objects via `owner<T*>` and for heap arrays via `owner<array_ptr<T,N>>` or `owner<span<T>>`.

In the `lifetime` profile, the following rules apply to `owner<T>`.

- `T` must be a non-owner indirection.
- An `owner<T>` obeys the following state transition table. Note: "valid" is spelled "`o'`".

| | **Pre** | **Post** |
|---|---|---|
| **DECLARE(o)** | (not applicable) | o is `invalid` |
| **o = new T** | o is `invalid` or `null` | o is o' |
| **o = o2** | o is `invalid` or `null`<br>(no requirement on o2) | o is o2's "pre" state<br>o2 is `invalid` |
| ***o**<br>**o->** | o is o' | (no change) |
| **&o** | o is in any state | (no change) |
| **delete o** | o is o' or `null` | o is `invalid` |
| **END_LIFE(o)** | o is `invalid` or `null` | (not applicable) |
| **ESCAPE(o)** | o is o' or `null` | (not applicable) |

- The following helpers to automate common usages to comply with the above table:

```
template<class T>
void move_owner(owner<T>& dst, owner<T>&& src)
    { delete dst; dst=src; src=nullptr; }

template<class T>
auto release_owner(owner<T>&& src) -> owner<T>
    { owner<T> tmp=src; src=nullptr; return tmp; }
```

- Constructing or assigning a `T` from an `owner<T> o` is like calling a smart pointer's "get" and yields a `T` with a lset of `{o'}`.
- Constructing or assigning an `owner<T>` from a `T` is not allowed.

**Notes** `owner<T*>` should be thought of as an unencapsulated `unique_ptr<T>`. That is, instead of performing ownership operations (such as moving ownership or performing `delete`) within encapsulated `unique_ptr` member functions, we expose the raw `*`, C++98-style, and require all callers to share the responsibility of collaboratively implement the same semantics by hand.

Constructing or assigning an `owner<T>` from a `T` is not allowed because it is unsafe. For example, code could copy the same pointer to two different owners, leading to eventual double `delete`.

For code that has owning raw pointers that intends unique ownership, change the variable's declaration from `T*` to `owner<T*>` and then any lurking mistakes will start to be diagnosed at build time.

### Shared ownership: `raw_shared_owner<>`

`raw_shared_owner<>` is a type alias that is intended for use primarily inside the implementation of smart pointers. `raw_shared_owner` is intended to be principally used for single heap objects via `raw_shared_owner<T*>` and for heap arrays via `raw_shared_owner<array_ptr<T,N>>` or `raw_shared_owner<span<T>>`.

**Note** The name is bikesheddable. This alias doesn't necessarily need to be used for shared ownership, but for any ownership convention that is not straight unique+`new`+`delete` ownership and so relies on surrounding code that we cannot check for correct enforcement. Because it's really "opaque" to the lifetime rules, it should have an ugly name. Potential alternative names include: `hidden_owner`, `manual_owner`, `opaque_owner`, `raw_opaque_owner`, `trustme_owner`, etc.

We are considering initially restricting `raw_shared_owner` to be a private data member, for use only in implementing non-unique owner library abstractions like `shared_ptr` and `gc_ptr`. However, if there are many places where it could be useful in existing code, we may need to allow its more widespread use despite its weaker guarantees.

In the `lifetime` profile, the following rules apply to `raw_shared_owner<T>`.

- `T` must be a non-owner indirection.
- A `raw_shared_owner<T>` obeys the following state transition table. Differences from the `owner<>` state transition tables are highlighted. Note that there is a new row to allow setting a `raw_shared_owner` from an `owner o`, where `__next` means the next invented shared name.

| | Pre | Post |
|---|---|---|
| **DECLARE(s)** | (not applicable) | `s` is `invalid` |
| **`s = o`** | `s` is in any state | `s` is `o`'s "pre" state,<br>replacing `o'` with `__next`<br>`o` is `invalid` |
| **`s = s2`** | `s` is in any state | `s` is `s2`'s "pre" state<br>(no change to `s2`) |
| **`*s`**<br>**`s->`** | `s` is `__name` | (no change) |
| **`&s`** | `s` is in any state | (no change) |
| **`delete s`** | `s` is `__name` or `null` | `s` is `invalid` |
| **END_LIFE(s)** | `s` is in any state | (not applicable) |
| **ESCAPE(s)** | `s` is `__name` or `null` | (not applicable) |

- Constructing or assigning a `T` from a `raw_shared_owner<T> s` is like calling a smart pointer's "get" and yields a `T` with lset `{s'}`.
- Constructing or assigning a `raw_shared_owner<T>` from a `T` is not allowed.

**Note** Unlike with `owner<>`, the `raw_shared_owner<>` rules do not by themselves guarantee freedom from leaks (failure to `delete`) or multiple deletion. Basic shared indirection types are intended to encapsulate a `raw_shared_owner` and then perform their own appropriate tracking, such as a reference count or tracing collection, to ensure that `delete` is called exactly once. However, because this responsibility is encapsulated within these few basic types, as long as their implementations are correct the entire program is correct.