# Default-Deny + Provable-Whitelist Invalidation

Safer Than Borrow-Checker in the Long Run


| | |
|---|---|
| Document #: | D4296R0 |
| Date: | 2026-07-14 |
| Project: | Programming Language C++ |
| Audience: | SG23 |


Reply-to: Sherry Ignatchenko <si@6it.dev>
Guy Davidson <gd@6it.dev>
Dmytro Ivanchykhin <di@6it.dev>
Marcos Bracco <mb@6it.dev>


## Contents

# 1 Abstract

Profile invalidation is one of the most critical (and most controversial) parts of the overall safety profiles effort. We'll describe our "default-deny + provable-whitelist" proposal for implementing profile invalidation. Very briefly, we aim to be *provably-safe* from the outset, while reducing the amount of false positives by adding "we can prove it is safe" rules aiming specific provably-safe patterns in existing C++ code. We're sure that in this way it is possible to achieve fewer false positives than with borrow-checker approaches, gradually approximating the "Holy Grail" of the safety profiles where all "good" code will be free from false positives. As an implementation detail of the provable-whitelist part of it, we want to exploit the existing C++ type system (including strict aliasing) to eliminate some of the common invalidations from the very beginning - and without the need to annotate.

# 2 Motivation

We feel that it is absolutely critical to:

— Have provable safety at all times
— Reduce the amount of false positives generated by perfectly good code as far as possible.

These two aims being in direct conflict creates a significant challenge.

# 3 Prior Art

We base our analysis mostly on P3446R0 ("profile-invalidation"), on P1179R1 ("lifetime-profiles"), on P3081R2 ("core safety profiles"; specifically, we're relying on the "type" profile and on the "bounds" profile), and on existing borrow-checker approaches including, but not limited to, P3390R0 ("Safe C++").

# 4 Disclaimer

All our musings below are currently still of a "thinking aloud" quality; beware of the potential mistakes in our claims and our reasoning. While we ourselves feel it is pretty solid, without public scrutiny we won't be able to say more than that. This is especially true given the sheer volume of this document; we're quite sure that we've made more than one mistake in the analysis below. On the other hand, we're sure that all such mistakes should be fixable within the general framework of this document, so any such mistakes should not affect the general premise we're presenting.

# 5 Principles

We wholeheartedly agree with all the Principles stated in P3446R0; restating them very briefly:

— we have to provide complete guarantees, wherever possible - by static analysis
— false-positives in static analysis are inevitable
— defaulting to safety
— static analysis should be local
— avoiding annotations as much as possible

In addition, we want to elaborate on the principles a little bit.

**AA.** From "complete guarantees" it follows that for static analysis, we are, in effect, dealing with proofs here (we want to *prove* that the program is safe).

**A.** Most importantly, we feel that

— minimizing the amount of false positives (that is, pieces of perfectly valid and well styled code) for which suppression of the profile is necessary is critical.

The thing is that even one single suppression drops all the guarantees for the whole program (that is, unless we establish manually-proven safety perimeters such as boundary of standard library). Additionally, interactions with the rest of the code mean that if we have one single suppression then, strictly speaking, changing ANY line of code in the whole codebase has a potential to break safety. Note that this problem of each suppression having potentially global effects is not unique to safety profiles. This applies, for example, to Rust's `{unsafe}` as well.

Because of it, we contend that

— **out of two systems each of which proves memory safety, the one which requires fewer suppressions, results in safer programs.**

**B.** We define the Holy Grail of the proving engine to be "If a senior code reviewer can see (staying within purely local analysis) that the function is safe, then our proof engine should see it too". This is consistent with P1179's 'human says "ok"' comments.

We feel that this approach is fundamentally safer (that is, can provide significantly fewer false positives) than engines derived from a single formal idea, such as a borrow-checker. At the very least we'll demonstrate an example which is not solvable by a borrow-checker, which a human being can prove as being perfectly safe purely locally.

**C.** As a corollary to "one single suppression drops all the guarantees for the whole program" observation, we feel that

— having suppressions within "safety perimeters" at library level are strongly preferable to accidental suppressions at application level.

The reasoning goes as follows: for rarely-changing library classes, proving their properties manually (by human code review etc.) is feasible, and after this manual proof is completed (which establishes a "safety perimeter"), the whole thing is safe. In contrast, for ad-hoc suppressions at application level, no such proofs are feasible, and as stated above, after having a single suppression beyond manually proven "safety perimeters", changing ANY line of code in the whole codebase can be unsafe.

**D.** We wholeheartedly agree with "Do not rely on non-local static analysis" principle, and moreover, we feel that this is a prerequisite to two other principles we want to introduce:

— decisions made by the analyzer have to be explainable to a human being
— code that is already safe should be maintainable without losing safety

Both these principles will get broken very quickly with non-local analysis. In other words, non-local analysis is not even a compiler performance issue: it is a much deeper problem of creating multi-page error messages, and worse still, if analysis across functions without annotations is performed, we'll be essentially relying on implementation details (which in turn will prevent refactoring). The spectre of making a minor change deep in implementation, which breaks the code using this function because some unannotated implementation detail was relied on 10 levels of calls higher, is looming over any substantially non-local analysis. This makes such approaches extremely fragile with respect to perfectly legitimate refactoring.

# 6 Definitions

## 6.1 Annotations are Provable

First of all, just to avoid any doubt, we treat all the annotations as *firm* and *provable* (for example, wherever we rely on an annotation in the callee, we *prove* the same annotation in the caller). In addition, whenever we have to suppress safety for an annotation to stand - such a suppression has semantics "developer has already proven it manually, and takes responsibility for it".

## 6.2 Containers and Pointers

We adopt definitions of "container" and "pointer" from P3274R0/P3446R0, "container" defined as "anything that can hold a value", and "pointer" defined as "any object that gives direct access to another".

More formally, we define "container" as any of:

— any class/struct with a raw pointer or another container within. *NB: it applies to all members, including private ones.*

— raw pointer `T*` (NB: here, we seem to differ from P3446R0); still, at the point when an `&p` is taken to make `T**` out of it, assignment to our `T*` can invalidate `T**`. As a result, we treat `T*` as both a container and a pointer to avoid special handling for `T**` etc. From a different angle: in a sense, from our current POV `T*` is equivalent to `struct { T* };`. Note though that it seems that raw reference (`T&`) is not a container (there is no way to invalidate this container as such).

We define "pointer" as any of:

— any class/struct with a raw pointer or raw reference (see below on raw references) or another poiner within (again, it applies even to private members). Note that this is very similar to our definition of "container", so most of the classes will be both pointers and containers at the same time. It should be possible, however, to annotate a class as `[[not_pointer]]`; this annotation should be available only under suppression, and is aimed to simplify/speed-up analysis for classes such as `std::unique_ptr<>`.

— raw pointer

— raw reference. While raw references cannot be invalidated in the sense that pointer-to-reference always stands, they can be invalidated in the sense that an object they're referring to, dies (goes out of scope etc.), so it *seems* that for our purposes, raw reference is a pointer - but is not a container.

For classes/structs-based pointers, there can be zero or more "assigning functions"; let's consider an example from P3446R0, section 3.1:

```
struct Holder{
  std::vector<int>* v_ = nullptr;
  void store(std::vector<int>& v) { v_ = &v; }
  void clear() { v_->clear(); }
};
```

Here, whenever we call `store()` - we're effectively assigning to our `struct Holder`; such calls should be treated as regular assignments by further analysis (in particular, by *psets* described below). We define the following way to find out functions (member or not) to be "assigning functions": "assigning function" for `class X` is any function which takes two pointers: 1st pointer being a non-const pointer to `class X` (in the example above - to `struct Holder`; for `store()` such a pointer is `this`), 2nd pointer being some non-const(?) pointer to some type `Y`. Note that this definition may be way too restrictive in certain cases, see Rule #3 which will allow to rule out some of spurious cases.

### 6.3 refcounted

On top of it, we introduce a concept of "refcounted", which is a basic reference-counted object. Note that `std::shared_ptr<>` is interpreted as refcounted + pointer (this is consistent with P1179R1, though our refcounted taken alone is narrower than P1179R1's SharedOwner).

## 7 Default-Deny + Provable-Whitelist Proposal

### 7.1 Philosophy

First of all, let's note that for the purposes of this current document, we consider only static analysis; generating runtime checks (as in for example, P3081R2's "as-if a contract_assert") are explicitly out of our scope. Note that in this particular document, we don't take a stance on "whether this or that UB (such as null pointer dereference), should be handled via static analysis (forcing changes such as inserting an explicit `if(p==nullptr)`, or a `[[not_null]]` annotation, to make code locally-UB-free) or by inserting runtime checks"; rather, we're demonstating that *if desired*, respective UBs *can* be handled using static analysis only.

We propose to deal with invalidation/lifecycles in the following manner:

**Layer 1. "Negative Baseline"**. First, looking at the N5046 (C++26 final draft), and at the P3596R2 ("Undefined Behavior and IFNDR Annexes"), we take relevant UBs from them (i.e. those UBs we want to

handle by invalidation-profile), and make an "Invalidation-Profile Negative Baseline" out of it; see section "List of UBs" below for the list of UBs we currently want to handle. *As soon as this "Negative Baseline" is established, proofread by multiple experts in the field, and the implementation enforces it - the invalidation-profile becomes provably-safe (with regards to those UBs it claims to handle).*

Note that at this point, we will NOT be looking to avoid false-positives; rather, the idea here is to ensure 100% provable safety. As a result, "Negative Baseline" is relatively simple. Let's also note that "Negative Baseline" should cover ALL P1179's 'human says "error"' cases; however, our logic is different from P1179 in that *we consider all the potential invalidations 'errors' by default*, so it is perfectly fine for the "Negative Baseline" to flag 'human says "ok"' cases too (they will be handled by "Positive Rules" discussed below).

**Layer 2. "Positive Rules"**. As a second layer, on top of this "Negative Baseline", we propose to introduce "positive" rules, each rule handling one locally-provable safe scenario. As a result, each rule will reduce the number of false-positives. An expected modus operandi would look roughly like this: somebody brings in a locally-provably-safe code example that is not handled by existing rules => it acts as a trigger for developing yet-another rule (or for improving an existing one without sacrificing provable-safety). Let's note that such rules correspond to P1179's 'human says "ok"' cases; more strictly, we can say that for such cases our logic is actually 'Error by default, but human says "ok"'.

Overall, our proposal is conceptually similar to "default-deny + provable-whitelist" approach used in security. Opposed to "default-permit" approaches (which are akin to playing whack-a-mole), "default-deny" is known to be extremely solid, and is often called a "gold standard" in security world.

Why this approach is safer than borrow-checker et al: because in the long run (as we develop more of those "Positive Rules") it will cause fewer false-positives; fewer false-positives, in turn, mean fewer suppressions, and having fewer suppressions, as discussed above, improves safety.

## 7.2 Negative Baseline: list of UBs

Here is the list of UBs which we proposed to address within invalidation profile. Wherever possible, we'll be using terminology from P3596R2.

— `[ub:lifetime.outside.*]` - we'll be erring on the safe side, and won't make allowances for specific still-valid cases; for the purposes of the Negative Baseline, ANY pointer is invalidated by mutation of ANY container, and ANY operation over invalidated pointer is flagged by "Negative Baseline". In addition, ANY call to a function passing a container or pointer by reference, is assumed to mutate. Of course, this will result in lots of false-positives, but as we described above, false-positives will be handled by "Positive Rules" described below. In addition, we'll be saying that within a safe program, all the pointers are valid ("valid" being defined either a pointer to a valid object or nullptr). It means that a safe function cannot possibly return a non-valid pointer (directly or indirectly; "indirectly" here means returning an object with non-valid pointer within); within our "err on safe side" model - we'll be prohibiting ALL the returns of the pointers (including assigning them to `T*&` function params), with "Positive Rules" allowing provably-safe scenarios. The same goes for throwing pointers (directly or indirectly) - it is prohibited by default.

— `[ub:original.type.implicit.destructor]` - we'll be prohibiting (flagging) placement-new and explicit destructor calls (that is, outside of suppressions); at least we'll be able to handle respective Example in P3596R2, and ATM, we don't know of other ways for a program to explicitly end the lifetime of an object.

*NB: from our experience, in the well-maintained code, trickery such as placement-new, explicit destructors, user-defined operator new, etc. belongs to specialized and tightly-controlled classes; within our concept of "safety perimeters", whole such classes are expected to be under suppression, making such class a "[manually proven] safety perimeter". This justifies our currently proposed approach of "just flagging them" (though there is a chance to isolate certain provably-safe patterns by adding "Positive Rules").*

— `[ub:creating.within.const.complete.obj]` - our understanding is that it should be handled by prohibiting placement-new described above.

— `[ub:basic.stc.alloc.dealloc.constraint]` - we'll be flagging all the user-defined operator new, operator new[], operator delete, operator delete[].

— [ub:intro.object.implicit.create] and [ub:intro.object.implicit.pointer] - handling TBD; does flagging malloc, free, memcpy()/memmove(), operator new, operator new[], operator delete, operator delete[], and casts such as reinterpret_cast<> prevent it? *NB: all these are banned/flagged as described below anyway.*

— [ub:basic.stc.alloc.zero.dereference] - "The effect of indirecting through a pointer returned from a request for zero size is undefined" - handling TBD

— [ub:basic.stc.alloc.dealloc.throw] - our understanding is that it should be handled by prohibiting operator delete/operator delete[] described above.

*NB: we consider [ub:basic.compound.invalid.pointer] a responsibility of bounds-profile*

— [ub:basic.start.main.exit.during.destruction] - we'll be flagging std::exit() within destructors.

— [ub:basic.start.term.use.after.destruction] - handling TBD; it *seems* to be sufficient to prohibit instantiation of classes which have constructors or destructors marked as [[may_use_global]] (see below), as objects having static storage duration.

— [ub:basic.start.term.signal.handler] - ??

— [ub:expr.unary.dereference] - to err on the safe side, we'll be prohibiting ALL the dereferences of pointers (with "Positive Rules" allowing dereferences if we can prove they're not null).

— [ub:expr.delete.mismatch] and [ub:expr.delete.array.mismatch] - we feel that we should have annotations such as [[owning_ptr]] T*, and [[owning_ptr]] std::span<T> (the latter - replacing those pointers which are allocated via new[] and should be deleted with delete[]; we feel it goes along the lines of bounds-profile). We'll prohibit delete on the pointers which are not annotated as [[owning_ptr]] ones. In addition, for all the [[owning_ptr]]-annotated pointers we'll *require* that delete/delete[] will be called on them (this is not to address UB, but is necessary to avoid memory leaks). In other respects, we'll consider [[owning_ptr]]-annotated entities as containers. *NB: We also feel that without an annotation/type of a pointer being an owning-pointer, provable safety won't be reachable (we just won't be able to distinguish a "normal" raw pointer from an "owning" one, at least for a function parameter).*

— [ub:expr.delete.dynamic.type.differ] and [ub:expr.delete.dynamic.array.dynamic.type.differ] - besides prohibiting ugly casts, we need to prohibit implicit casts-to-base for [[owning_ptr]].

*NB: [ub:expr.mptr.oper.not.contain.member] seems to be handled by type-profile (by prohibiting casts)*

— [ub:expr.mptr.oper.member.func.null] - we'll prohibit function calls via pointer to member (that is, unless one of "Positive Rules" can prove that the pointer is not null).

— [ub:stmt.return.flow.off] and [ub:stmt.return.coroutine.flow.off]- we'll prohibit "flowing off" for non-void functions / coroutines (for coroutines it should be based on semantics) *NB: we're not sure if it belongs to invalidation-profile, so if any other profile will handle it, we're perfectly fine with it.*

— [ub:class.dtor.no.longer.exists] - should be handled by the same logic as described for [ub:original.type.implicit.destructor]

— [ub:class.base.init.mem.fun] - handling TBD; is it sufficient to flag all calls of member functions within base constructor param expressions, pre-conditions of constructors, and post-conditions of destructors?

— [ub:class.cdtor.before.ctor], [ub:class.cdtor.after.dtor], [ub:class.cdtor.convert.pointer], and [ub:class.cdtor.form.pointer] - handling TBD; is it sufficient to flag (in addition to prohibiting explicit destructors and placement-new) (a) all the pointers to members of global objects (taken directly from the object), (b) taking pointers to members and calling non-static member functions within constructor initializer lists, and (c) virtual inheritance? *NB: as always for "Negative Baseline", these restrictions can be relaxed by "positive rules"; now we're merely establishing a "provably-safe baseline".*

— [ub:class.cdtor.virtual.not.x], [ub:class.cdtor.typeid], and [ub:class.cdtor.dynamic.cast] - handling TBD; prohibiting typeid()/dynamic_cast<>/virtual calls within constructors/destructors

should be sufficient, but potentially transitive nature of it may require annotations (will we have to annotate all the member functions callable from constructors a-la `[[callable_from_constructor]]`, and prohibit `typeid()`, `dynamic_cast<>`, and virtual calls in such member functions?)

— `[ub:except.handle.handler.ctor.dtor]` - handling TBD, but it *seems* sufficient to prohibit (a) direct access to members, and (b) calls of non-static member functions in function-try-block of constructor/destructor.

*NB: While the following are not UBs, we still want to address them*

— we want to handle use of the object after being moved-from; while technically, the behavior is not UB, but "unspecified", this "unspecified" induces lots of highly-dependent-on-specifics UBs. To prevent them, we propose to consider "use-after-being-moved-from" (besides calling a destructor on the moved-from object) as a *potential UB* for the purposes of our "Negative Baseline".

— with regards to *refcounted*, we want to prevent memory leaks due to runtime cycles; however, as we're restricted to static analysis, we propose to prohibit *refcounted* cycles at type level (i.e. `class A` having `refcounted<B>` is ok only as long as there is no `refcounted<A>` within `class B`, directly or indirectly), which will in turn guarantee that there are no runtime cycles which cause memory leaks. Note that lambdas should be considered as types for this purpose (so runtime cycles created via `std::function<>` will be prohibited too), and `std::any<>` should be considered as a `refcounted<*>`. Handling of caveats related to cycles created by polymorphic classes is TBD, but for starters, we can consider polymorphic class as another `refcounted<*>`.

IMPORTANT: we understand that the list of UBs above is extremely ambitious; and we're sure that we made more than a few mistakes in our very preliminary analysis above. However, OTOH we're sure that none of those mistakes is fatal, and that all of them can be fixed (in particular, by addressing criticisms from the community). In other words, ATM we do not claim that we have SOLVED all these UBs; what we claim is that these UBs are SOLVEABLE.

In addition, most likely, there are quite a few more invalidation-related issues in C++, but we hope they should be handleable along the same lines as those above.

## 7.3 Prerequisite Annotations

We consider the following antemnotations as a prerequisite; note that while we are relying on them, we DO feel that each of them simply annotates certain practice:

### 7.3.1 `[[semantic_const]]` or `[[deep_const]]`.

It seems to us that, formally, we cannot guarantee that if we call a `const` member function of a container, invalidation can't possibly occur. Indeed, if container contains `string*` inside, no existing formality prevents writing a function which is technically `const`, but which assigns to `string[0]`, invalidating pointers to `char*` in the outside world. Of course, well-behaving containers don't do it, but distinguishing well-behaving ones from ill-written ones is not that easy. To deal with it, it seems to be necessary to introduce a `[[semantic_const]]` annotation for container member functions or an even more general `[[deep_const]]` annotation for the pointers.

— Default: while it would be nice to have already-existing `const` imply `[[deep_const]]`, this would probably break backward compatibility way too much, so for the time being we're considering adding `[[deep_const]]` (or `[[semantic_const]]`) as an additional attribute.

### 7.3.2 `[[may_use_static]]` or (better) `[[may_use_static(type)]]`

The idea here is to say that all the functions which use objects with static object duration, MUST be annotated as such. In addition to being used for the purposes of the proving engine, we feel that documenting such behaviors will improve code readability; such functions are not usual in modern code, and any side effects they may cause, are largely unexpected. Note that `thread_local` will need either its own annotation, or should fall under the same `[[may_use_static]]` (this depends largely on concurrency-profile). Let's further note that Meyer's singletons will also have to be annotated; this is perfectly consistent with C++ Core Guidelines' "Singletons are basically complicated global objects in disguise.".

— Default: We feel that defaulting to (i.e. not annotating) "good" functions (those which don't use global/static data) will cause many fewer annotations across the code base.

— Let's note that a conspicuous absence of such an annotation will provide solid proof of "`int* f(int* a, int* b)` strictly meaning that (in the absence of pointer arithmetic) returned `T*` is either `a`, or `b`" (as casts are already banned, it just doesn't have anywhere else to come from). In other words, P1179R1's "by default we assume that a function returns values that are derived from its arguments" becomes a much stronger "we can FORMALLY PROVE that function returns values that are derived from its arguments" (sic!).

— As a side benefit, such annotations (or an actionable lack of them) will most likely allow compilers to perform certain optimizations too (while it is not an exact match for GCC/Clang "`pure`" or "`const`" functions, it still provides valuable information about guarantees provided by the function).

#### 7.3.3 `[[may_invalidate]]`

an annotation along the lines of P3446R0's `[[invalidating]]`. Consistently with P3446R0, by default, functions are not allowed to call mutating functions on its container parameters; to enable such mutations - `[[may_invalidate]]` is required.

Note that these are only most important annotations; other annotations will be introduced in appropriate contexts below.

### 7.4 Implementation Prerequisite: Explicitation

In C++ code, it is common to have certain things implicit; of our particular interest are temporaries, and guaranteed (or not guaranteed) order of evaluation according to the standard. At the stage of explicitation, we want to rewrite the code to the form which makes all such things explicit (to avoid any doubt - it is an automated rewriting, not hand-performed). For example,

```
f(x+y);
```

could be rewritten into

```
auto __tmp1234 = x+y;
f(__tmp1234);
[[explicitation::kill(__tmp1234)]];
```

And

```
f(g(x),h(y));
```

could be rewritten into something along the lines of

```
[[explicitation::begin_unordered]]
auto __tmp1235 = g(x);
auto __tmp1236 = h(y);
[[explicitation::end_unordered]]
f(__tmp1235,__tmp1236);
[[explicitation::kill(__tmp1235)]];
[[explicitation::kill(__tmp1236)]];
```

Note that we do not mean that the rewrite above stands for all possible function declarations of `f`, `g`, and `h`; rather, we mean it it as an illustration that such rewrite is certainly possible and well-defined by the standard in each particular case (even though order of certain calculations can remain unspecified by the standard, as illustrated by `begin_unordered`/`end_unordered` annotations).

While it is just a purely technical step, it will help splitting our elephant task into more digestible (and debuggable) parts, separating concerns of "what does standard say about lifetimes and sequences" and pure formal logic of the dependencies and lifetime analysis.

### 7.5 Implementation Prerequisite: Flow Analysis (psets)

We propose to base implementation of *psets* described in detail in P1179R1.

To quote P1179R1: "For each local pointer variable, track the set of things it could point to at any given point in the function's source code, by following the function's control flow paths. In this paper, that is called the *pset*, or "points-to" set, which typically contains the name of local variables, but can also contain the special values *null*, *global*, and *invalid*."

To accommodate P1179R1's *psets* for our current purposes, quite a few changes will be necessary, but we're sure that such changes will be minor and won't cause any drastic changes to our further analysis. A few known differences include:

— we differ with P1179R1 in our analysis of passing a container which has only move semantics (such as `unique_ptr<>`, or `[[owning_ptr]] T*`) to a function as parameter by value. While P1179R1 treats it via considering `std::move()` as a function which invalidates the pointers to container - we prefer to be more explicit, bringing the object to "moved-from" state - it will allow a tad better ground for further "positive rules", as well as a bit better error messages.

— unlike P1179R1, we can formally prove "that a function returns values that are derived from its arguments"; the difference lies with us introducing `[[may_use_static]]`, and actionable absence of it (without such an annotation/actionable lack of annotation, formal proof of "derived from its arguments" is not possible, as the function may return a pointer to static or global, so our formalism seems to be different from P1179R1's here). Note that for more complicated types, proofs will become more difficult.

— we stipulate that "assigning functions" (see "Definitions" section) are treated similar to regular pointer assignments; however, "assigning functions" are to be treated as "potential assignments", so respective *pset* is not replaced by new value, but is rather *expanded*.

— along the lines of P3446R0, we'll need to support `[[not_returned(p)]]` (where `p` is an expression based on function params)

### 7.5.1 csets: psets for containers

For our analysis, we need a construct which is similar to *psets* (in a sense that it is a set of potential attributes which correspond to some portion of the code), but applies to containers; we name it *cset*. For the time being, the only attribute which we need to handle for containers, is whether the container in question was already moved-from. It is not clear to us whether P1179R1 implied it (in particular, in an Example in 2.4.5.3), but in any case we feel necessary to articulate it.

In addition, as a part of cset, for each container we maintain its "origin"; "origin" is a descriptor of the "where this container variable may have came from". For example, if container variable came from input parameter, we say its origin is "`param1`" ("`param2`" etc.). If it was created as a local variable - we say its origin is "`local1`" ("`local2`" etc.). And if it was created on heap (such as via `new()` or via `make_unique()`) - we says its origin is "`heap1`" ("`heap2`" etc.). Note that there can be multiple origins for the same container variable, and that they are subject to the set math similar to that of the *psets* (for example, if in one branch origin is {"`local1`"}, and in another branch origin is {"`param1`"}, then after merging these branches, we get {"`local1`","`param1`"}).

### 7.5.2 "path constraints" as a potential extension of pset

Let's further note that, when/if it becomes desirable, it is possible to come with an extension of the P1179R1's *psets*, extending them to full-scale "path constraints" (as described in [HorvathEtAl], and used by at least Clang Static Analyzer). OTOH, at least ATM, we don't advocate for doing it (agreeing with P3446R0's "It would be unfortunate if correctness varied with the cleverness of implementers" argument, at least until we can formalize "path constraints" sufficiently).

[HorvathEtAl] Gabor Horvath, Reka Kovacs, Zoltan Porkolab, *Scaling Symbolic Execution to Large Software Systems, https://arxiv.org/pdf/2408.01909*

## 7.6 Positive Rules

Now, let's get to our "Positive Rules"; as noted above, each of them should be driven by a code example.

### 7.6.1 Rule #0: Unrelated Types don't Interact

First of all, as noted above, we want to exploit the C++ type system, with our base intuitive example being that "`vector<string>.push_back()` should not invalidate any `int*`" (which is consistent with our "if a senior reviewer can see it" paradigm):

```
void f(std::vector<std::string>>& v, int* p) {
  v.push_back("");//does NOT invalidate p
  //...
}
```

The basic idea is that if two types are completely unrelated (in the extreme case - know absolutely nothing about each other) invalidation cannot possibly happen without any understanding of `v`'s and `p`'s origins - merely based on the *types* involved.

Further, let's note that such unrelated types is an extremely common pattern; *lots* of classes out there have absolutely nothing to do with each other, and we feel that exploiting this knowledge to reduce the number of false positives (and number of otherwise-required annotations) wherever possible is a good idea.

Overall, we feel that type-based analysis is a critical part of the efforts to reduce the number of false positives for invalidation. Or looking at it from a different angle, C++ is traditionally very much type-oriented, so missing an opportunity to exploit its type system would be a waste.

While strict formalism is still TBD, our current feeling is that it should be sufficient that:

1. The container being mutated (let's name it `class X`), doesn't have any data members (including non-public!) which are of type `Y*` (where `Y*` is a candidate for invalidation), AND
2. `class X` does not have any members (including non-public!) of type `Y` either, AND
3. The mutating function being called is NOT annotated with any of `[[may_use_static]]` annotations (see above), AND
4. The mutating function being called isn't passed any pointers of type `Y*`.

Actually, strictly speaking, we can say that condition #1 is a special case of condition #4, applied to the `this` pointer which is implicitly passed to member functions.

Note that conditions 1, 2, and 4 are transitive - i.e. if `X` contains `Y*` where `Y` contains `Z*`, mutating `X` can invalidate `Z*`; also, "`Y*`" includes pointers to any classes which are derived from `Y` (so if `X` contains `Y1*` where `Y1` is derived from `Y`, mutating `X` can invalidate `Y*`).

The basic idea for the correctness proof goes as follows: "*if some class `X` doesn't have access to any pointers to `Y`, then in the absence of casts, which are already banned by type profile per P3081R2, it cannot possibly create non-null `Y*` out of nowhere*".

All of it may sound complicated, but (a) in fact, it is just formalizing basic human intuition (such as that `vector<string>` has nothing to do with `int*`), and (b) indeed, `vector<string>.push_back()` cannot possibly invalidate `int*`, *never ever, regardless of context (sic!)*. Everything else is merely a formalization of this basic human intuition.

Other examples solved: in a whole bunch of RL cases (depending on the nature of the classes `Logger` and `Widget`), Rule #0 will be able to prove correctness of `Logger` and `Widget` example from section 1.8 of P3446R0 without any annotations. Indeed, if `Widget` and `Logger` are unrelated, their instances cannot possibly interact in some implicit manner. NB: sure, Rule #0 won't remove the need for the annotation described in P3446R0's 1.8 *in the general case*, but removing such a need in a whole bunch of real life cases, is already a good thing (per our Principles).

#### 7.6.1.1 Discussion

It can be argued that this way we'll be relying on implementation details of our classes, and relying on implementation details, as discussed above, is a non-starter because of fragility and preventing refactoring. However, if we look at it more closely, we'll find that while in a general case, relying on implementation details is indeed a taboo, in this particular case, it is neither fragile nor does it prevent refactoring. We argue that in practice, this very specific kind of inter-class relationship is extremely stable with regards to sensible refactorings, and while each individual pointer is indeed an implementation detail, the "knowledge of"

category of inter-class relations is *very much* related to the essence of the class, and as a result are extremely stable across refactorings. With our "litmus test" example of "mutating `vector<string>` shouldn't invalidate any `int*`" we cannot see any feasible implementation which would need an `int*` within (unless casts are used, but under our no-cast assumption using casts is equivalent to suppression anyway). As an illustration of our logic here, while `vector<T>` can be legitimately implemented in half a dozen different ways (as 3x `T*`, as `T*` +2x offsets, etc.), this "`vector<T>` contains `T*` and knows of no other types and therefore can invalidate pointers to `T` and ONLY pointers to `T`" stays *invariant* with regards to all these implementations of `vector<T>`. Moreover, we contend that it stays *invariant* with regards to any reasonable implementations which avoids using `reinterpret_cast`. We expect the same to stand across a vast majority of real life classes, which makes relying on such analysis to be not overly-fragile with respect to legitimate refactorings.

Also let's note that this aspect of Rule #0 is similar to our definition of "container" and "pointer" - we have to define both of them in terms of having pointers or references inside, even if these pointers or references are private.

#### 7.6.1.2 Bottom line on Rule #0

Based on our analysis above, we feel that our Rule #0 will result in a significant reduction of false positives or need to annotate (compared to the comparable proving systems without type analysis). In particular, it does help to solve P3446R0's example of `Logger` and `Widget` (and to solve it trivially, without annotations), as well as our litmus-test case of `vector<string>.push_back()` (not) invalidating `int*`.

Also, let's note that application of our Rule #0 corresponds to the proof in the worst-case when both entities involved (one is a container being modified, and the other is a pointer being invalidated) are pointers/references which came as unannotated function parameters. In such cases, we know absolutely nothing about them, so they can easily alias, unless aliasing is proven to be impossible by type analysis/Rule #0.

As a side note, we feel that this whole invalidation task is at least conceptually similar to the well-known problem of "potential pointer aliasing".

### 7.6.2 Rule #1: Patently Independent Containers don't Interact

"Rule #1" stems from the need to handle the following example:

```
vector<Element> process_elements(list<Element>& elem) {
    vector<Element> result;
    bool progress = true;
    while (progress) {
        progress = false;
        for (auto iter = elem.begin(); iter != elem.end();) {
            if(can_process(iter)) {
                progress = true;
                result.push_back(iter); //(1)
                iter = elem.erase(iter); //(2)
            }
            else
                ++iter;
        }
    }
    return result;
}
```

This seems to be perfectly valid code, which can be understood by our "senior reviewer", so we certainly want to handle it. However, our "Rule #0" won't be able to prove that mutation in line (1) can't possibly invalidate `iter`, so we will have to flag use of `iter` in line (2). To avoid this flagging, we propose to go along the following lines (which are reminiscent of the Lifetime Profile (P1179R1)).

First, let's say that we have "proven binding of a pointer to a container" at some point in code, if and only if *pset* of this pointer has exactly one element which corresponds to the container, at this point in code.

Now, as we defined this "proven binding", we can say that IF the pointer is "proven to be bound" to a specific container at the point of mutating operation, it is NOT invalidated by this mutating operation on a

patently-different container.

"patently-different" here is about being able to prove that two containers cannot possibly alias. This proof of containers being patently-different can be done either (a) on the basis of container types (`list<int>` cannot alias with `vector<int>`, exact formalism TBD, but is expected to be along the same lines as "Rule #0"), or (b) on the basis of proving that physical addresses of the containers involved are different. This second (b) analysis is done based on "origin" part of our *csets*. In particular, "`local*`" cannot possibly alias with "`param*`" and with "`heap*`", "`param*` cannot possibly alias with "`heap*`", different "`local*`" origins cannot possibly alias with each other, and different "`heap*`" origins cannot possibly alias with each other. However, different "`param*`" origins can alias with each other - that is, unless they're annotated with `[[not-aliased-with(param_name)]]`. Such origin-based analysis is *provably-correct* on the one hand, and allows to identify that containers are patently-different (cannot possibly alias) wherever possible. Note that this "origin" approach seems to handle correctly `shared_ptr<>`s too.

Yet another way to prove lack of container aliasing is to rely on annotations (as always, such annotations have to be provable in the context of the caller, or they will be flagged).

Now, applying this logic to the example above: First, `iter` is always bound to `elem`, and second, we can prove that `elem` and `result` containers cannot possibly alias with each other.

Actually, in this particular example we can prove the second part *twice*: first, by type analysis (we don't see how `list<T>` can possibly alias with `vector<T>` - that is, unless `T` has vectors or lists respectively), and second, by "one being `param` and another being `local`" (using *cset* "origins"). It means that even if both `elem` and `result` are lists (or vectors), we still can prove they're guaranteed not to alias based on the "origins". On the other hand, if both `elem` and `result` come as function parameters, but are of incompatible types (such as `list<T>` and `vector<T>`), we still can prove they're different because of their types being different.

Therefore, as we have proven both that `iter` is always bound to `elem`, and that `elem` and `result` cannot possibly alias, `result.push_back()` cannot possibly invalidate `iter`.

As a side benefit, such a "proven to be bound" analysis can be used to rule out issues with a mismatching iterator and container (for example, as in "if we cannot prove an iterator is bound to a container, any passing of the iterator to the container's member functions is unsafe"). While (as we understand) this is out of scope of the invalidation profile, it should be necessary within some other safety profile.

Side note: in the example above, `iter=elem.erase(iter)` seems to be ok just because parameters get evaluated before the function is called, so `erase(iter)` is legal; and while `erase(iter)` does invalidate `iter`, after the call to `erase(iter)`, `iter` is not used, just assigned, which is fine. More formally, sequencing guarantees are to be handled by our Explicitation step, and by the time we reach Rules, most of the complications will be already handled (though we'll still need to account for `[[explicitation::unordered_*]]` blocks mentioned in the Explicitation section above).

Another (even more side) note: we don't think that ensuring that dealing with potentially iterating past list's `end()` belongs to invalidate profile (it looks more like a bounds profile's responsibility), but wherever necessary, it can be handled along the same lines too (for example, via checking that after each increment and after each `iter=erase(iter)`, we check for `end()` and terminate the loop. This seems to be sufficient to prove the correctness of not going out of bounds for lists etc.).

### 7.6.3 Rule #2: Returning Pointers to Statics and Input Params is Ok

Per Negative Baseline above, to be on a safer side, we ban returning ANY pointer from a function. However, per "Rule #2", if a pointer is known to point either to an object with a static object duration, or to input parameter (="has *pset* which has ONLY 'global', or input param"), it is ok to return it (as long as return is NOT marked as `[[owning_ptr]]`). Also, it is ok to return an `[[owning_ptr]]` if return value is guaranteed to be annotated `[[owning_ptr]]`.

Example (from P3446R0, section 1.5); our added comments are marked with `IDIB::`

```
int* glob = nullptr;
int* f(int* p)
{
  return p; // OK; IDIB: Ok per Rule #2
  return glob; // OK; IDIB: Ok per Rule #2
```

```
  return new int{7}; // OK (but that has different problems);
                     // IDIB: flagged per Negative Baseline;
                     // would be ok if return value is marked as [[owning_ptr]]
}
```

In a somewhat-similar manner, we allow throwing pointers to 'global' object, and `[[owning_ptr]]` pointers (this includes allowing throwing objects which contain them); how to prevent memory leaks when throwing bare `[[owning_ptr]]` pointers is TBD.

#### 7.6.4 Rule #3: Incompatible Types Don't Count as Assigning Functions

In Definitions, we gave the following definition of "assigning function": ""assigning function" for `class X` is any function which takes `two` pointers: 1st pointer being a non-const pointer to `class X`, 2nd pointer being some non-const(?) pointer to some type `Y`."

Example we want to handle is modified example from P3446R0, section 3.1:

```
struct Holder{
  std::vector<int>* v_ = nullptr;
  void print(std::string& v) { /* some formatting into std::string */ }
  void clear() { v_->clear(); }
};
```

Here, we can *prove* that if `class X` does NOT have (directly or indirectly) any members which are assignable a pointer to `Y` (or any pointer reachable from `Y`), there is no chance that `class X` can be modified to carry any pointer which is a part of pointer to `Y`. Hence, we don't need to consider such functions as "assigning functions". Note that this analysis (just like with "Rule #0") is purely type-based, with absolutely no analysis of the function body.

#### 7.6.5 Rule #4: Dereferencing provably non-null pointers is fine

Rule #4 says that that "if we can prove that pointer is non-null - it is ok to dereferece it". Such proofs can either rely on `[[not_null]]` pointer annotations (as all our annotations, such annotations will have to be proven themselves), or on flow analysis along the lines of P1179R1; for example,

```
void f(T* p) {
  if( p )
    //nullptr is excluded from pset(p) here
    *p = 42;//ok
}
```

# 8 Pros and Cons of Proposed Approach

## 8.1 Pros

Formal safety correctness (of course, modulo bugs both in proofs and in implementations). In each and every case, we *prove* that invalidation is outright impossible. This provides a major advantage over P1179R1; in a sense, we're trying to *upgrade* P1179R1's logic to cover *all* the cases within particular classes of UB's.

On the other hand, we managed to avoid forcing one single programming paradigm down the throats of C++ developers. It *seems* to us that we managed to navigate between the Scylla of non-provable safety (as in P1179) and the Charybdis of requiring the rewriting tons of code which is both perfectly safe and perfectly good according to modern C++ guidelines (as in P3390R0) (while annotating will be necessary, we distinguish between "rewriting" and "annotating").

Proposed approach is patently local with regards to function calls. No implementation change to a function 10 levels below or above it can possibly lead to a "Houston, we've had a problem" scenario.

While being local, we don't rely *solely* on stack analysis. As is shown by real-world use of purely stack driven models such as a borrow-checker, they tend to effectively force suppression in way too many places across code (which in turn drops safety guarantees). In contrast, this proposal exploits advantages of the C++

type system (in particular, strict aliasing). Indeed, we insist that pushing into `vector<char>` should not invalidate any of `int*` just because of the way `vector<char>` is designed.

Gradual handling of more and more practical use cases while staying within the same framework (and while staying 100% safe at each and every point). If the number of "Positive Rules" is large enough, our approach can cover pretty much ANYTHING which falls under our "if a senior developer can see it." paradigm. This is a very serious advantage, both in light of false positives being highly undesirable, and from the point of view of being better (actually, "safer", as each case where we don't need suppression makes the program safer) than competing programming languages.

Note that taken to the extreme, one of the "Positive Rules" can even prove correctness (which BTW our "senior reviewer" can see too) of the following:

```
vector<int> v{1};
v.reserve(2);
auto p = &v[0];
v.push_back(2);//no reallocation possible due to earlier v.reserve()
               //=> we can prove that no invalidation happens
int x = *p;//still ok
```

By the way, this is one example which (as far as we understand) is patently impossible to address (without suppression, that is) within purely stack-driven provers such as borrow-checker.

More generally, we're sure that in the long run (=given enough of "Positive Rules") our approach will become *substantially better* (both in terms of app-level suppressions, and in terms of being able to recognize perfectly-safe existing code) than a borrow-checker one. This, per our Principle A, means that **in the long run, our approach will result in safer programs than borrow-checker alone can possibly provide.**

Potential for **annotations-only** changes to most of existing demonstrably-safe code. Note that annotations-only changes may be preferred during code hardening of huge existing code bases (*it is much easier to demonstrate that code changes which only add static-analysis annotations, are perfectly safe*, and review only real safety fixes manually). While on this way current proposal won't be sufficient (in particular, most likely, statically-provable pre/post-conditions will be necessary), we're positive about handling *vast majority* of the existing code in a statically-safe manner (with only a handful of contracts being enforced in runtime). Let's also that making all safety-profiles annotations-only will require rather ugly annotations which will allow to emulate `std::span<>` over two parameters (such as `T*` and `size_t`), but if there is such a demand - it can be done too.

Ability to explain our reasoning. We'll be able to explain that "line #456 was flagged because `p` was invalidated in line #123", and that "mutation of type `X` in line 123 can invalidate `Y*` (which is then used in line #456) because `X` has `Z* f()`, and `Z` has `Y0* g()`, and `Y` is derived from `Y0`", and that "containers `elem` and `result` can alias because their types are compatible, and because they both come as unannotated function parameters". Note that, for "Positive Rules" rules we'll have to choose which Rules to explain (as in "if the Rule is specific to list-like containers, don't mention it unless a list-like object is involved"), but overall, it doesn't look too bad.

Overall, we feel that this is a reasonable way to eat safety-profiles elephant, one bite at a time.

## 8.2 Cons

Need to introduce the `[[may_use_static]]` annotations. While we argue that it is a good thing to have such annotations anyway, the most strong argument for them being documenting the code much better, some programming styles (which rely on globals everywhere, and which we argue to be long-obsolete as of 2026) will still see it as a burden.

Formal reliance on class implementation details (such as non-public data members). While above we argue that in real life such inter-class relations are much more fundamental than a mere implementation choice (see `Logger`-vs-`Widget` and `vector<char>`-vs-`int`), and that therefore, we won't get maintenance-time fragility with respect to sensible refactorings, this claim is still to be validated.

Difficult to standardize due to sheer volume of those "Positive Rules". A mitigating factor is that standardization can be done gradually too.

# 9 Appendix. Examples Handled

In the Appendix, we'll describe how our proposal would handle examples from various papers (and some of our own ones too). Our added comments will start with `IDIB:`

It is interesting to note that ALL the specially-designed relevant Examples were covered just by 5 "Positive Rules"; while we certainly do NOT claim that they will be sufficient to cover all RL cases, but we feel it indicates that overall number of Rules needed to cover 99% of "good" code won't be in hundreds.

## 9.1 Relevant Examples From P3596R2

From p.60 (illustrating `[ub:intro.object.implicit.create]`):

```
void f()
{
  void *p = malloc(sizeof(int) + sizeof(float));
  *reinterpret_cast<int*>(p) = 0;
  *reinterpret_cast<float*>(p) = 0.0f; // undefined behavior, cannot create
                                       // both int and float in same place
}
```

— each `reinterpret_cast<>` will be flagged.

From p.61 (illustrating `[ub:intro.object.implicit.pointer]`):

```
#include <cstdlib>
struct X {
  int a, b;
  ~X() = delete; // deleted destructor makes X a non-implicit-lifetime class
};
X* make_x() {
  // The call to std::malloc can not implicitly create an object of type X
  // because X is not an implicit-lifetime class.
  X* p = (X*)std::malloc(sizeof(struct X));
  p->a = 1; // undefined behavior, no set of objects give us defined behavior
  return p;
}
```

— C-style cast `(X*)` will be flagged.

```
#include <cstdlib>
struct X {
  int a;
};
struct Y {
  int x;
};
*make_y() {//IDIB: is there a typo?
  X* p1 = (X*)std::malloc(sizeof(struct X));
  p1->a = 1;
  Y* p2 = (Y*)p1;
  p2->x = 2; // undefined behavior, lifetime of p1 was started but not
             // ended and lifetime of p2 was not started.
  return p2;
}
```

— C-style casts `(X*)` and `(Y*)` will be flagged.

From p.20 (illustrating `[ub:lifetime.outside.*]`):

```
#include <cstdlib>
struct B {
  virtual void f();
```

```
  void mutate();
  virtual~B();
};
struct D1 : B { void f(); };
struct D2 : B { void f(); };
void B::mutate() {
  new (this) D2; // reuses storage - ends the lifetime of *this
  f(); // undefined behavior
  ... = this; // OK, this points to valid memory
}
void g() {
  void* p = std::malloc(sizeof(D1) + sizeof(D2));
  B* pb = new (p) D1;
  pb->mutate();
  *pb;
  void* q = pb;
  pb->f();
}
```

— placement `new` will be flagged.

From p.62 (illustrating [ub:lifetime.outside.pointer.delete]):

```
struct S {
  float f = 0;
  ~S() {}
};
void f() {
  S* p = new S;
  p->~S();
  delete p; // undefined behavior, operand of delete, lifetime has ended and S
            // has a non-trivial destructor
}
```

— explicit destructor call will be flagged.

From p.62 (illustrating [ub:lifetime.outside.pointer.member]):

```
struct S {
  float f = 0;
};
float f() {
  S s;
  S* p = &s;
  s.~S();
  return p->f; // undefined behavior, accessing non-static data member after
               // end of lifetime
}
```

— explicit destructor call will be flagged.

From p.63 (illustrating [ub:lifetime.outside.pointer.virtual]):

```
struct B {};
struct D : virtual B {};
void f() {
  D d;
  D* p = &d;
  d.~D();
  B* b = p; // undefined behavior
}
```

— explicit destructor call will be flagged.

From p.63 (illustrating [ub:lifetime.outside.pointer.dynamic.cast]):

```
struct B { virtual~B() = default; };
struct D : B {};
void f()
{
  D d;
  B* bp = &d;
  d.~D();
  D* dp = dynamic_cast<D*>(bp); // undefined behavior
}
```

— explicit destructor call will be flagged.

From p.63 (illustrating [ub:lifetime.outside.pointer.glvalue.access]):

```
void f() {
  int x = int{10};
  using T = int;
  x.~T();
  int y = x; // undefined behavior, glvalue used to access the
  // object after the lifetime has ended
}
```

— explicit destructor call will be flagged.

From p.63 (illustrating [ub:lifetime.outside.pointer.glvalue.member]):

```
struct A {
  void f() {}
};
void f() {
  A a;
  a.~A();
  a.f(); // undefined behavior, glvalue used to access a
         // non-static member function after the lifetime has ended
}
```

— explicit destructor call will be flagged.

From p.64 (illustrating [ub:lifetime.outside.pointer.glvalue.virtual]):

```
struct B {};
  struct D : virtual B {
};
void f() {
  D d;
  d.~D();
  B& b = d; // undefined behavior
}
```

— explicit destructor call will be flagged.

From p.64 (illustrating [ub:lifetime.outside.pointer.glvalue.dynamic.cast]):

```
struct B { virtual~B(); };
struct D : virtual B {};
void f() {
  D d;
  B& br = d;
  d.~D();
  D& dr = dynamic_cast<D&>(br); // undefined behavior
}
```

— explicit destructor call will be flagged.

From p.64 (illustrating [ub:original.type.implicit.destructor]):

```
class T {};
struct B {
  ~B();
};
void h() {
  B b;
  new (&b) T;
} // undefined behavior at block exit
```

— placement `new` will be flagged.

From p.64 (illustrating [ub:creating.within.const.complete.obj]):

```
struct B {
  B();
  ~B();
};
const B b;
  void h() {
  b.~B();
  new (const_cast<B*>(&b)) const B; // undefined behavior
}
```

— placement `new` will be flagged.

From p.65 (illustrating [ub:basic.stc.alloc.dealloc.constraint]):

```
#include <new>
void* operator new(std::size_t sz) {
  if (sz == 0)
  return nullptr; // undefined behavior, should return non-null pointer
  return std::malloc(1); // undefined behavior, if successful
                         // should return allocation with
                         // length in bytes is at least as large as the requested size
}
void operator delete(void* ptr) noexcept {
  throw 0; // undefined behavior, terminates by throwing an exception
}
```

— user-defined `operator new` and `operator delete` will be flagged.

From p.66 (illustrating [ub:basic.stc.alloc.dealloc.constraint]):

```
struct X {
  void operator delete(void*) { throw "oops"; }
};
void f()
{
  X* x = new X();
  delete x; // undefined behavior
}
```

— user-defined `operator delete` will be flagged.

From p.22 (illustrating [ub:creating.within.const.complete.obj]):

```
struct B {
  B();
  ~B();
```

```
};
const B b;
void h() {
  b.~B();
  new (const_cast<B*>(&b)) const B; // undefined behavior
}
```

— explicit call to destructor and placement `new` will be flagged.

From p.64 (illustrating [ub:creating.within.const.complete.obj]):

```
struct B {
  B();
  ~B();
};
const B b;
void h() {
  b.~B();
  new (const_cast<B*>(&b)) const B; // undefined behavior
}
```

— explicit call to destructor and placement `new` will be flagged.

From p.66 (illustrating [ub:basic.stc.alloc.zero.dereference]):

```
void test()
{
  char* c = static_cast<char*>(operator new(0z));
  c[0] ='X'; // undefined behavior
}
```

— `static_cast` will be flagged.

From p.66 (illustrating [ub:basic.stc.alloc.dealloc.throw]):

```
struct X {
  void operator delete(void*) { throw "oops"; }
};
void f()
{
  X* x = new X();
  delete x; // undefined behavior
}
```

— user-defined `operator delete` will be flagged.

From p.68 (illustrating [ub:basic.start.main.exit.during.destruction]):

```
#include <cstdlib>
struct Exiter {
  ~Exiter() { std::exit(0); }
};
Exiter ex;
int main() {}
// undefined behavior when destructor of static variable ex
// is called it will call std::exit
```

— `std::exit()` within destructor will be flagged.

From p.69 (illustrating [ub:basic.start.term.use.after.destruction]):

```
struct A {};
[[may_use_global]] void f() {//IDIB: [[may_use_global]]
                             //is required to have static within
```

```
  static A a;
}
struct B {
  B() { f(); }//IDIB: flagged as a call of [[may_use_global]]
};
struct C {
  ~C() { f(); }//IDIB: flagged as a call of [[may_use_global]]
};
C c;
B b; // call to ‘f()‘ in constructor begins lifetime of a
int main() {}
// undefined behavior, static objects are destructed in reverse order,
// in this case a then b and finally c. When the destructor of c is called,
// it calls f() which passes through the definition of previously destroyed
// block-scope object
```

From p.70 (illustrating [ub:expr.unary.dereference]):

```
int f()
{
  int *p = nullptr;
  return *p; // undefined behavior
}
```

— dereference flagged by default (see comment in Negative Baseline section).

From p.77 (illustrating [ub:expr.delete.mismatch]):

```
[[owning_ptr]] int* x = new int;//IDIB: [[owning_ptr]] is required
delete[] x; // undefined behavior, allocated using single object new expression
```

— `delete[]` flagged as mismatching (`[[owning_ptr]] std::span<>` is required for `delete[]`).

From p.77 (illustrating [ub:expr.delete.dynamic.type.differ]):

```
struct B {
  int a;
};
struct D : public B {
  int b;
};
void f() {
  [[owning_ptr]] B* b = new D;//IDIB: cast from [[owning_ptr]] D*
                             //      to [[owning_ptr]] B* will be flagged
  delete b; // undefined behavior, no virtual destructor
}
```

From p.77 (illustrating [ub:expr.delete.dynamic.array.dynamic.type.differ]):

```
struct B {
  virtual ~B();
  void operator delete[](void*, std::size_t);
};
struct D : B {
  void operator delete[](void*, std::size_t);
};
void f(int i) {
  D* dp = new D[i];
  delete[] dp; // uses D::operator delete[](void*, std::size_t)
  B* bp = new D[i];
  delete[] bp; // undefined behavior
}
```

— user-defined `operator delete` will be flagged.

From p.78 (illustrating [ub:expr.mptr.oper.member.func.null]):

```
struct S {
  int f();
};
void f() {
  S cs;
  int (S::*pm)() = nullptr;
  (cs.*pm)(); // undefined behavior, the second operand is null
}
```

— call via pointer to member will be flagged.

From p.81 (illustrating [ub:stmt.return.flow.off]):

```
int f(int x) {
  if (x)
    return 1;
// undefined behavior if x is 0
}
void b() {
}
```

— lack of return will be flagged.

From p.86 (illustrating [ub:class.dtor.no.longer.exists]):

```
struct A {
  ~A() {}
};
int main() {
  A a;
  a.~A();
} // undefined behavior, lifetime of a already ended before implicit destructor
```

— explicit call to destructor will be flagged.

From p.47 (illustrating [ub:class.base.init.mem.fun]):

```
class A {
  public:
  A(int);
};
class B : public A {
  int j;
public:
  int f();
  B() : A(f()), j(f()) { } };
class C {
  public:
  C(int);
};
class D : public B, C {
  int i;
  public:
  D() : C(f()), i(f()) { }
};
```

— calls of member function `f()` within constructor param expression will be flagged.

From p.87 (illustrating [ub:class.base.init.mem.fun]):

```
class A {
  public:
  A(int);
};
class B : public A {
  int j;
  public:
  int f();
  B()
  : A(f()), j(f()) {} // undefined: calls member function but base A not yet initialized
// well-defined: bases are all initialized
};
class C {
  public:
  C(int);
};
class D : public B, C {
  int i;
  public:
  D()
  : C(f()), i(f()) {} // undefined: calls member function
                      // but base C not yet initialized
  // well-defined: bases are all initialized
};
```

— calls of member function `f()` within constructor param expression will be flagged.

From p.48 (illustrating [ub:class.cdtor.before.ctor]), same example on p.88:

```
struct X { int i; };
struct Y : X { Y(); }; struct A { int a; };
struct B : public A { int j; Y y; }; // non-trivial
// non-trivial
extern B bobj;
B* pb = &bobj; int* p1 = &bobj.a; int* p2 = &bobj.y.i; // OK; IDIB: flagged
// undefined behavior: refers to base class member
// undefined behavior: refers to member's member
A* pa = &bobj; B bobj; // undefined behavior: upcast to a base class type
                       //IDIB: flagged
// definition of bobj
extern X xobj;
int* p3 = &xobj.i; X xobj; //IDIB: flagged
// OK, all constructors of X are trivial
For another example,
struct W { int j; };
struct X : public virtual W { };//IDIB: flagged virtual inheritance
  struct Y {
  int* p;
  X x;
  Y() : p(&x.j) { }
  // undefined, x is not yet constructed
};
```

— wherever we flag OK scenarios, they can be fixed by adding Positive Rules.

From p.89 (illustrating [ub:class.cdtor.after.dtor]):

```
struct X {
  int i;
  ~X(); // non-trivial
```

```
};
X& g()
{
  static X x;
  return x;
}
void f()
{
  X* px = &g();
  px->~X();
  int j = px->i; // undefined behavior
}
```

— explicit call to destructor will be flagged.

From p.49 (illustrating [ub:class.cdtor.convert.pointer] and [ub:class.cdtor.form.pointer]):

```
struct A { };
struct B : virtual A { };
struct C : B { };
struct D : virtual A { D(A*); };
struct X { X(A*); };
struct E : C, D, X {
E() : D(this), X(this) {} // undefined behavior: upcast from E* to A* might use path E* →D* →A*
// but D is not constructed
// "D((C*)this) " would be defined: E* →C* is defined because E() has started,
// and C* →A* is defined because C is fully constructed
// defined: upon construction of X, C/B/D/A sublattice is fully constructed
};
```

— virtual inheritance will be flagged.

From p.89 (illustrating [ub:class.cdtor.form.pointer]):

```
struct A {
  int i = 0;
};
struct B {
  int *p;
  A a;
  B() : p(&a.i) {} // undefined behavior
};
```

— taking pointer to member in constructor initializer list will be flagged.

From p. 49 (illustrating [ub:class.cdtor.virtual.not.x]), same example on p.90:

```
struct V {
  virtual void f();
  virtual void g();
};
struct A : virtual V {
  virtual void f();
  struct B : virtual V {
  virtual void g();
  B(V*, A*);
};
struct D : A, B {
  virtual void f();
  virtual void g();
  D() : B((A*)this, this) { }
};
```

```
B::B(V* v, A* a) {
  f(); g(); v->g(); a->f(); // calls V::f, not A::f
  // calls B::g, not D::g
  // v is base of B, the call is well-defined, calls B::g
  // undefined behavior: a's type not a base of B
}
```

— virtual inheritance, as well as cast, will be flagged.

From p.90 (illustrating [ub:class.cdtor.typeid]):

```
struct V {
virtual void f();
};
struct A : virtual V {};
struct B : virtual V {
  B(V *, A *);
};
struct D : A, B {
  D() : B((A *)this, this) {}
};
B::B(V *v, A *a) {
  typeid(*this); // std::type_info for B
  typeid(*v); typeid(*a); // well-defined: *v has type V,
    // a base of B yields std::type_info for B
    // undefined behavior: type A not a base of B
}
```

— virtual inheritance and typeids in constructor will be flagged.

From p.50 (illustrating [ub:class.cdtor.dynamic.cast]):

```
struct V {
  virtual void f();
};
struct A : virtual V { };
struct B : virtual V {
  B(V*, A*);
};
struct D : A, B {
  D() : B((A*)this, this) { }
};
B::B(V* v, A* a) {
  typeid(*this); // type_info for B
  typeid(*v); // well-defined: *v has type V, a base of B yields type_info for B
  typeid(*a); // undefined behavior: type A not a base of B
  dynamic_cast<B*>(v); // well-defined: v of type V*, V base of B results in B*
  dynamic_cast<B*>(a); // undefined behavior: a has type A*, A not a base of B
}
```

— virtual inheritance, and each typeid/dynamic_cast in constructor will be flagged.

From p.91 (illustrating [ub:class.cdtor.dynamic.cast]):

```
struct V {
  virtual void f();
};
struct A : virtual V {};
struct B : virtual V {
  B(V *, A *);
};
struct D : A, B {
```

```
  D() : B((A *)this, this) {}
};
B::B(V *v, A *a) {
  dynamic_cast<B *>(v); // well-defined: v of type V*, V base of B results in B*
  dynamic_cast<B *>(a); // undefined behavior: a has type A*, A not a base of B
}
```

— virtual inheritance and each `dynamic_cast` in constructor will be flagged.

From p.92 (illustrating [ub:except.handle.handler.ctor.dtor]):

```
#include <iostream>
struct A {
  A() try : x(0 ? 1 : throw 1) {
  } catch (int) {
    std::cout << "y: " << y << std::endl; // undefined behavior, referring to non-  static member y in
  // the handler of function-try-block
  }
  int x;
  int y = 42;
};
```

— access to non-static `y` in function-try-catch block of constructor will be flagged.

## 9.2 ALL Examples From P3446R0

From 1.2:

```
void f(int x)
{
  [[owning_ptr]] int* p = new int{7};//IDIB: per Negative Baseline, we'll insist on annotating owning
  [[owning_ptr]] int* q = new int{8};
  delete q;//IDIB: ok but invalidates q per Negative Baseline
  *q = 9; // likely disaster (disallow); IDIB: flagged as q is invalidated
  if (x) *q = 11; // possible disaster (disallow); IDIB: same as above
  if (x) delete p;//IDIB: invalidates p per Negative Baseline
  *p = 10; // likely disaster (disallow); IDIB: flagged as p is potentially invalidated
  if (0<x) *p = 1; // possible disaster (disallow); IDIB: same as above
  //IDIB: we'll flag that not all branches deleted an [[owning_ptr]] p, preventing memory leak here
}
```

```
void use(int* p) // p is valid because it must be valid in the calling scope
{
  // ...
  delete p; // possible disaster (disallow); IDIB: flagged as p is not annotated as [[owning_ptr]]
}
void f(int*p)
{
  use(p);
  *p = 7; // disaster; IDIB: impossible if use(p) doesn't have p annotated as [[owning_ptr]]
          //           IDIB: OTOH, if use(p) _is_ annotated as [[owning_ptr]], use(p) above will inval
}
```

From 1.3:

```
void f()
{
  int* p = nullptr;
  int* q = nullptr;
  {
    int x = 7;
```

```
    p = &x;
    // ...
    if (x) q = &x;
  }//IDIB: invalidates both p and q (x gets out of scope, both p's and q's psets include x)
  *p = 9; // likely disaster (disallow); IDIB: flagged as p is invalidated
  *q = 7; // likely disaster (disallow); IDIB: same as above
}
```

```
int* g()
{
  int x = 7;
  int y = 9;
  // ...
  if (x) return &x; // likely disaster (disallow); IDIB: flagged per Negative Baseline (ANY returned p
  return &y; // likely disaster (disallow); IDIB: same as above
}
```

From 1.5: see Rule #2 above

From 1.6:

```
std::span<int> make_span(std::vector<int>& v)
{
  return std::span<int>(v);
}
std::span<int> foo()
{
  std::vector<int> v{1,2,3,4,5};
  auto span = make_span(v);
  return span; // escaping pointer (disallow); IDIB: flagged per Negative Baseline
}
```

```
std::span<int> foo(std::vector<int>& v)
{
  auto span = make_span(v);//IDIB: just from make_span() signature, we can prove that span points with
  return span; // escaping pointer (allow; IDIB: allowed per Rule #2 above
}
```

```
std::vector<int> glob{1,2,3,4,5};
std::span<int> foo(std::vector<int>& vv) [[may_use_static(std::vector<int>)]] //IDIB: we force [[may_u
{
  auto span = make_span(glob);
  return span; // escaping pointer (allow); IDIB: allowed per Rule #2
}
```

```
std::span<int> foo(std::vector<int>& vv) [[may_use_static(std::vector<int>)]] //IDIB: we force [[may_u
{
  auto span = make_span(glob);
  return span; // escaping pointer (allow); IDIB: allowed per Rule #2
}

int* glob() [[may_use_static(int)]] //IDIB: we still force [[may_use_static]]; in our world, it is a g
{
  static int x = 7;
  return &x;
}
```

```
int* min(int*,int*); // return pointer to the smallest int pointed to
int x = 7;
int check(int* p)
{
```

```
  int y = 9;
  return min(&x,&y); // disallowed; IDIB: disallowed per Negative Baseline; Rule #2 doesn't apply as p
}
```

From 1.8:

This is an elaborated example involving `Logger` and `Widget`. As discussed above, in many of the RL cases (depending on the nature of these classes, in particular, when they're unrelated) we'll be able to prove its safety using our "Rule #0". In a *general* case, however, an annotation (along the lines suggested by P3446R0) `[[not_returned(this->logger)]]` will be necessary.

From 1.10:

```
struct B {
};
struct D : B {
  int* p;
};
[[owning_ptr]] B* f()//IDIB: we force [[owning_ptr]] to support returning new pointers
{
  int xx;
  D* dp = new D{{},&xx};//IDIB: original says new D{7}, but new D{{},&xx} is the best we can make out
  return dp; // pointer to local pointer tries to escape (disallow);
             //IDIB: we have to deal with 2 pointers which are effectively returned here: dp and dp->p
             //IDIB: while for dp returning new is ok (per Rule #2), for dp->p Negative Baseline appli
}
```

From 2:

```
void g()
{
  vector<int> vi { 1,2 };
  auto p = vi.begin(); // point to first element of vi
  vi.push_back(9); // may relocate vi's elements (invalidating p)//IDIB: Negative Baseline invalidates
  *p = 7; // likely disaster (disallow); IDIB: flagged as p is invalidated
}
```

From 2.1:

```
void f(vector<int>& vi [[may_invalidate]])//IDIB: we prefer [[may_invalidate]] name for this annotatio
{
  vi.push_back(9); // may relocate vi's elements
}
void g()
{
  vector<int> vi { 1,2 };
  auto p = vi.begin(); // point to first element of vi
  f(vi);//IDIB: as param is [[may_invalidate]], it does invalidate pointers, including p
  *p = 7; // likely disaster (disallow); IDIB: flagged as p is invalidated
}
```

```
void f(vector<int>& vi) // the common default use, disallowing invalidation
{
  vi[2] =7; // not const, but OK
  vi.push_back(9); // may relocate vi's elements (disallow); IDIB: flagged as [[may_invalidate]] is mi
}
```

From 2.2:

```
int x = 7;
int* p = &x;//IDIB: per our Definitions, we consider p as both container and pointer
int** pp = &p;
cout << **pp; // 7
```

```
p = nullptr;//IDIB: container p is mutated, invalidating all pointers per Negative Baseline
cout << **pp;// likely disaster(disallow); IDIB: flagged as pp is invalidated above
```

Our own modification of it:

```
int x = 7;
int* p = &x;//IDIB: per our Definitions, we consider p as both container and pointer
int* p1 = &x;
int** pp = &p;
int** pp1 = &p1;
cout << **pp; // 7
p = nullptr;//IDIB: container p is mutated, invalidating all pointers per Negative Baseline
            //      However, pp1 is not invalidated per Rule #1 (p1 and p are patently different conta
cout << **pp;//IDIB: flagged as pp is invalidated above
cout << **pp1;//IDIB: not flagged as pp1 was not invalidated
```

From 3:

```
void use([[owning_ptr]] int* p)//IDIB: purely syntactic change
{
  // ...
  delete p; // this delete is required; IDIB: indeed required, per Negative Baseline
}
void f()
{
  [[owning_ptr]] int* p = new int{7};
  use(p);//IDIB: invalidates p per Negative Baseline
  *p = 7; // disallow: we know that p has been deleted; IDIB: flagged as p is invalidated
}
```

```
class Int_vec {//IDIB: ATM, we suggest suppression over the whole Int_vec
  int* elem;
  int sz;
  int_vec(int s) : elem{new int[s]}, sz{s} {}
  ~int_vec() { delete[] elem; }
};
```

From 4:

```
[[owning_ptr]] int* p = new int{7};
int* q = p; // create an alias
delete p;//IDIB: mutating container p; invalidates all the pointers including q (per Negative Baseline
*q = 9;//IDIB: flagged as q is invalidated
```

```
int* f(int* p) { return p; }
[[owning_ptr]] int* p = new int{7};
int* q = f(p); // create an alias; IDIB: we can derive that q is p,
               //        IDIB: but even with such a limited pset for q, it becomes a pointer to contai
delete p;//IDIB: mutating container, invalidating all pointers including q
*q = 9;//IDIB: flagged as q is invalidated
```

```
int* f(int* p) { return p; }
auto p = make_unique(7);
int* q = f(p.get()); // create an alias; IDIB: this is a pointer bound to container p
*q = 9;//IDIB: ok, no invalidation has happened
```

From 3.1:

```
struct Holder{
  std::vector<int>* v_ = nullptr;
  void store(std::vector<int>& v) { v_ = &v; }//IDIB: we consider this function as an "assigning funct
  void clear() { v_->clear(); }
```

```
};
void foo()
{
  std::vector<int> v{1,2,3,4,5};
  Holder inv;//IDIB: Holder is both container and pointer
  inv.store_vector(v); // inv becomes an alias for v; IDIB: calling an "assigning function" causes pse
  // ...
  inv.clear(); // not const: assumed to invalidate inv (disallow); IDIB: mutating inv, invalidating al
  v[1] =7; // v has been invalidated; IDIB: flagging as v was invalidated
}
```

```
template<typename T1, typename T2>
struct ZippedIterator {
  std::vector<T1>::iterator it1;
  std::vector<T2>::iterator it2;
};
template<typename T1, typename T2>
  auto zip(std::vector<T1>& v1, std::vector<T2>& v2) {
    //IDIB: we can prove that zip() returns iterator to either v1 or v2 (based solely on its declarati
  return ZippedIterator<T1, T2>{ v1.begin(), v2.begin() };
}
std::vector<int>::iterator inner(std::vector<int>& v) {
  std::vector<int> v2{6,7,8,9,10};
  auto it1 = zip(v, v2); // returns a local and a non-local
  return it1; // may return a pointer to the local v2 (disallow); IDIB: flagged by Negative Baseline;
  //IDIB: if, however, v2 would be an input parameter, Rule #2 would apply, allowing to return it1
}
void outer()
{
  std::vector<int> v{1,2,3,4,5};
  auto it = inner(v);
}
```

## 9.3 ALL examples from P1179R1

*NB: we did leave 3 of examples out of our current scope (all of them being lib-related; if this paper will get traction, we plan to handle those 3 examples in a separate paper dedicated to libraries and safety perimeters in general).*

From 1.1.1:

```
int* p = nullptr;
{
  int x = 0;
  p = &x; // A
  *p = 42; // B: human says "ok"; IDIB: no invalidation in sight, ok
} // C; IDIB: x goes out of scope, pset(p) = {invalid}
*p = 42; // D: human says "error"; IDIB: flagged due to pset
```

For several further examples (all the way to 1.1.4) - our analysis is identical to that of P1179R1 (after all, we're basing our reasoning on *psets*).

From 1.1.4:

In the following example, it all depends on "what are those `get_a_value()` and `get_another_value()` functions do":

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

```
}
```

If (as we assume) both `get_a_value()` and `get_another_value()` are annotated as `[[may_use_static(string)]]` - we'll allow this code. Indeed, taken in isolation, this code is safe - nothing bad can happen here (and formally, we'll apply Rule #2 to allow `return local;`).

OTOH, we'll handle concern of P1179R1 about the following code

```
string_view s = foo("x","y");
string_view s2 = foo("x","y"); // could silently invalidate s; IDIB: invalidation of all the pointers
print(s); // s could now be dangling; IDIB: flagging due to invalidation in the previous line
```

within this second code snippet. As `s` and `s2` are pointers, and call `foo("x","y")` is a call of a function which is annotated as `[[may_use_static(of non-const value)]]` - such as call effectively has an implicit non-const parameter of type `string`. As a result, invalidation is in order at the point of 2nd call of `foo()` (and Rule #1 won't be able to prove that `s` and `foo()` apply to patently different containers).

In the following example, it is assumed that `foo()` is NOT using any statics, and is NOT annotated:

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

From our standpoint, we are back to the P1179R1's reasoning (temporaries which went out of scope), aided with our Explicitation step.

Examples from 2.4.2.1 , 2.4.3.1, 2.4.4, and 2.4.5.1 are again handled consistently with P1179R1.

From 2.4.5.2:

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

```
void consume(vector<int>&&);
void test() {
vector<int> v(1000);
auto iter = v.begin(); // pset(iter) = {v'}
consume(std::move(v)); // A: pset(iter) = {invalid}, pset(v) = {invalid}; IDIB: we mark cset(v) as {mo
*iter; // error, iter was invalidated on line A
v[100]; // error, v is moved-from and [] has a precondition
        //  IDIB: to stay on a safer side, we prohibit ALL the accesses after move; if necessary, we c
        //        which would allow calling functions annotated with something like [[ok_moved_from]]
}
```

Examples from 2.4.6, 2.4.6.1, 2.4.6.2, 2.4.6.3, 2.4.7.1, 2.4.7.2, 2.4.7.3, 2.4.8.1, 2.4.8.2, 2.4.9.1, 2.4.9.2, and 2.4.9.3 are again handled consistently with P1179R1.

From 2.4.9.4:

```
p = &a[0]; // pset(p) = {a}
bool must_delete = false;
for( /*...whatever...*/ ) {
  // ...
  if( /*...whatever...*/ ) {
    // ...
    p = new A // A: pset(p) = {temp'}
        ;     // KILL(temp) → pset(p) = {invalid}
              // ERROR: no delete of owner<> returned from new
              // IDIB: we simply disallow such an assignment if p is not annotated with [[owning_ptr]]
```

```
    must_delete = true;
    // ...
  }
  // ...
}
if(must_delete)
  delete p; // ERROR, delete of non-owner<> is not lifetime-safe
            // IDIB: in our case, delete must be unconditional for all the pointers
            //       marked with [[owning_ptr]]
```

While mechanics here is different from that of P1179R1's, end-result here is the same: there should be two pointers (and whether 2nd pointer is `unique_ptr<>` as in P1179R1's next example, or whether it is `[[owning_ptr]] T*`, it doesn't matter for safety). This is one of the examples where perfectly-valid code cannot be salvaged by mere annotations (so if there are other suggestions how it can be handled, they're very welcome); fortunately, this kind of trickery is reasonably rare, and rewrite is straightforward.

Example from 2.4.10.1 is again handled consistently with P1179R1.

From 2.4.10.2:

```
static int gi = 0;
void f() {
  int i = 0;
  throw &i; // ERROR; IDIB: prohibited by Negative Baseline
  throw &gi; // OK; IDIB: salvaged by Rule #2
}
```

Example from 2.5.7.2 is again handled consistently with P1179R1.

Example from 2.5.7.3 is again handled consistently with P1179R1 (taking into account that, as we understand, P1179R1's `[[lifetime_const]]` is equivalent to P3446R0's conspicuous (and actionable!) absence of `[[invalidating]]` (which in turn corresponds to our conspicuous-and-actionable absence of `[[may_invalidate]]`).

Example from 2.5.7.4 is again handled consistently with P1179R1.

From 2.5.7.5:

```
widget& get_widget() [[may_use_static(widget)]] { // pset(ret) = {global};
                     // IDIB: we require [[may_use_static]] annotation here
  static widget w;
  return w; // ok, {global} is substitutable for {global}; IDIB: ok per Rule #2
}
```

Examples from 2.5.7.6 and 2.5.7.7 are again handled consistently with P1179R1.

From 2.5.7.8:

```
int* f() [[may_use_static(int)]]; // pset(ret) = {global}
         // IDIB: we require [[may_use_static]] annotation here, and (like P1179R1)
         //       infer that return data has pset = {global} (we can strictly prove it)
int main() {
  int* p = f(); // pset(p) = {global}
  *p = 42; // ok; IDIB: ok per Rule #2
}
```

From 2.5.7.9:

```
void g(shared_ptr<int>& s [[may_invalidate]], int* p) { // pset(s) = {s'}, pset(p) = {p}
  s = something_else; // KILL(s') → no local effect, does not kill p
                      // IDIB: alongside with P3446R0, we do NOT allow mutations
                      //       unless s is annotated; so for the whole example to
                      //       stand, s MUST be annotated as [[may_invalidate]]
  // ... // pset(p) == {p}, unchanged
```

```
}
// At call sites
// IDIB: the rest of the example is handled along the lines of P1179R1
int gi = 0;
shared_ptr<int> gsp = make_shared<int>();
int main() {
  // passing global and local objects
  f(&gi); // ok, pset(arg) == {global}
  int i = 0;
  f(&i); // ok, pset(arg) == {i}
  f(gsp.get()); // ERROR (lifetime.3), pset(arg) == {gsp'}, gsp is global
  auto sp = gsp;
  f(sp.get()); // ok, pset(arg) == {sp'}, sp's address has not been taken
  g(sp, sp.get()); // ERROR (lifetime.3), pset(arg2) == {sp'}, sp being passed
  g(gsp, sp.get()); // ok, pset(arg2) == {sp'}, sp not being passed
}
```

*As for examples in P1179R1's 2.5.7.10 and 2.5.7.11 - we don't include them in our current scope; overall, analysis of interactions with containers and special functions in std:: lib is a separate huge topic, which should be addressed separately (see our note in Rule #1 which gives a hope in this regard, but which annotations are the best at the lib level to describe the restrictions, and especially whether the Rules should be aware of lib-specific requirements and guarantees, is still TBD).*

Example from 2.5.7.12 is again handled consistently with P1179R1 (with the same notes as mentioned with respect to 2.5.7.3 above).

Examples from 2.6.1, 2.6.2.1, 2.6.2.2, and 2.6.2.3 are again handled consistently with P1179R1.

From 2.6.2.4:

```
struct X { int a, b; };
int& f(X& x) {
  return x.a; // ok, pset(ret) == pset(x)
  // IDIB: salvaged by Rule #2
}
```

Examples from 2.6.2.5, 2.6.2.6, and 2.6.2.7 are once again handled consistently with P1179R1.

Example from 2.6.3.1 is out of our scope (it involves a 3rd-party lib which we're not experts with).

Example from 2.6.3.2 is once again handled consistently with P1179R1.

From 2.6.3.3:

```
template <typename T>
class Expected {
  T& get() { return value_; }
  T value_;
  // ...
};
void take_it(Baz&&);
Expected<Baz> result = baz();
do_some_work(result.get()); // pset(tmp) = {result'}
take_it(std::move(result.get()); // A: added recently in a rush
                                 // pset(tmp) = {result'}, passed to rref param,
                                 // → KILL(result') and pset(result) = {invalid}
do_other_work(result.get()); // ERROR (lifetime.1): 'result' was invalidated when
                             // its data was moved from (line A)
                             // IDIB: flagging it as cset(result) has {move-from} in it
```

## 9.4 Our Own Examples

Disclaimer: examples in this section are AI-generated.

```
#include <iostream>
#include <memory>
#include <utility>
int main() {
    auto p = std::make_unique<int>(42);
    auto q = std::move(p);
    // UB: p is now nullptr after the move.
    std::cout << *p << '\n';//(a)
}
```

— `std::move` will mark `cset(p)` as moved-from, so its use in line (a) will be flagged.

```
#include <iostream>
#include <memory>
struct B;  // Forward declaration
struct A {
    std::shared_ptr<B> b;
    ~A() {
        std::cout << "A destroyed\n";
    }
};
struct B {
    std::shared_ptr<A> a;
    ~B() {
        std::cout << "B destroyed\n";
    }
};
int main() {
    auto a = std::make_shared<A>();
    auto b = std::make_shared<B>();
    a->b = b;
    b->a = a;
    std::cout << "Leaving main...\n";
}
```

— circular dependency of `shared_ptr<>s` between `class A` and `class B` (which may cause memory leak illustrated in `main()`) will be flagged.