P3442R1 – [[invalidate_dereferencing]] attribute

Document number: P3442R1

Date: 2025-01-13

Authors:

- Patrice Roy: patricer@gmail.com
- Nicolas Fleury: nidoizo@gmail.com

Reply to: patricer@gmail.com

Target audience: SG23, LEWG, SG14

Revision history

R0 to R1: SG23 had requested that consideration would be given to the harmonization of P3442 with P3465 - Pursue P1179 as a Lifetime Safety TS with which there was a potential overlap. See Integration with similar work for details.

Introduction

Some functions take arguments that offer indirect access to objects and invalidate that object as a result of their execution. We propose an attribute, `[[invalidate_dereferencing]]`, that would allow one to convey that information in the source code and help compiler provide better diagnostics.

This paper is part of the set of requests that can be found in P2996.

Motivation

Consider:

```
// ...
auto p = std::malloc(sizeof(T));
// ...
// fill the sizeof(T) bytes of buffer p with values that
// represent a T, perhaps with data read from a file
// ...
T * q = std::start_lifetime_as<T>(p);
// ... use q ...
q->~T(); // finalize the T object
std::free(p); // deallocate the underlying storage
// past this point, using *q or *p is incorrect, but
// mentioning this is QoI. Standard libraries and compilers
// might have tools to produce diagnostics to that effect
```

We propose to allow annotating an argument that would be erroneous to dereference past the point where it was used with `[[invalidate_dereferencing]]` to inform compilers of this fact.

Intended usage

Taking a example the function above, the signature for `std::free()` would be, if using this new attribute:

```
namespace std {
   std::free([[invalidate_dereferencing]] void *);
}
```

Note: `std::free()` in this example is just that: an example. This proposal does not suggest that standard library functions should use this new attribute, leaving such decisions to quality of implementation.

Effect

The expected effect of this attribute would be for a compiler to emit diagnostics when a function argument is used through a dereferencing operation such as the unary `*` operator (including the array subscript operator) or the `->` operator the after having been passed as argument to a function when that argument is annotated with `[[invalidate_dereferencing]]`, unless that argument is a pointer and has been assigned a new value. For example:

```
void *allocate(std::size_t);
void* reallocate([[invalidate_dereferencing]] void*,
                 std::size_t new_size);
void deallocate([[invalidate_dereferencing]] void*);
struct X { void f(); /* ... */ };
void test() {
   X *p = static_cast<X*>(allocate(10 * sizeof(X)));
   // ... can use *p, p-> or p[i] here...
   for(int i = 0; i != 10; ++i) p[i].f();
   p = static_cast<X*>(reallocate(p, 20 * sizeof(X)));
   // ... can use *p, p-> or p[] here
   X *q = static_cast<X*>(reallocate(p, 20 * sizeof(X)));
   // ... diagnostic expected if *p, p-> or p[] used here
   deallocate(q);
   // ... diagnostic expected if *q, q-> or q[] used here
}
```

Prior art

At least one standard library vendor (the Microsoft STL implementation) annotates arguments with `_Post_invalid_` to achieve effects analogous to those described here and suggested for the `[[invalidate_dereferencing]]` attribute.

Integration with similar work

During the 2024 Wrocław meeting, SG23 raised concerns about the integration of this proposal with the effort outlined in P3465 – Pursue P1179 as a Lifetime Safety TS.

The authors of P3442 and P3465 had private discussions on this topic:

- According to the author of P3465, C-style API types (owning raw pointer parameters, and owning handles like Win32 HANDLE that are actually integers) are not covered in P1179 so in that case, P3442 could be a compatible extension
- There is a possible overlap on owner types such as smart pointers and containers that already have first-class treatment in P1179. In the words of the author of P3465: "just passing a smart pointer by reference to non-const is already recognized by P1179 as a mutating operation that by default invalidates the Owner smart pointer (no annotation required) and automatically invalidates any local raw pointers/references that could refer to the object owned by the smart pointer.
- The author of P3465 adds "Functions that take a smart pointer by reference to non-const but never change its value can be annotated [[lifetime_const]] to say otherwise, but that seems odd for smart pointers. That makes more sense for non-owning containers like std::map where you know a non-const function like insert doesn't actually invalidate anything)"

It seems, thus, that [[invalidate_dereferencing]] remains useful. Cases to consider include:

- Owning arguments that do not convey ownership semantics through their type (types like HANDLE and raw pointers that are owners in C-style APIs come to mind).
- Smart pointers that are non-standard and might not benefit from the "no annotation" approach that covers standard containers and smart pointers.

The author of P3465 suggests however that the names of the attributes could be harmonized. Something like [[owner_destroyed]] has been mentioned as this would be closer to the terminology used in P1179. This paper makes no claim that [[invalidate_dereferencing]] is the best name for this feature; if the idea is adopted in the standard and the proposed semantics are deemed satisfying, we can discuss a different name if the one currently proposed is deemed perfectible.

FAQ

Question 00: have you considered alternative spellings?

Answer: not for now, but if the feature is accepted and the name poses a problem then we can discuss alternatives. Let's work with the current name for the moment to keep things simple.

Question 01: is `[[invalidate_dereferencing]]` intended to be used elsewhere than on function arguments?

Answer: not for now. This can be reconsidered if a convincing argument is made.

Question 02: is the use of an object annotated as invalidated through `[[invalidate_dereferencing]]` expected to produce a warning or an error?

Answer: the intent is to have a warning, but the end result is expected to be QoI as this sort of behavior can be impacted by such things as compiler settings.

Question 03: is the effect of [[invalidate_dereferencing]] expected to escape the context in which the invalidation occurs? Is it expected to be reported for potential invalidation cases? Both are expected to be QoI. Example:

```
void *allocate(std::size_t);

void deallocate([[invalidate_dereferencing]] void*);

// ...

struct X { /* ... */ };

template <class T> void g(X *p, bool b) {

   if(b) {

      deallocate(p); // note: invalidates p

      // p->f(); // diagnostic expected here

   }

   // possible diagnostic of potential use after

   // invalidation (QoI)

   p->f();

}

X* f(bool mystery) {

   X *p = static_cast<X*>(allocate(sizeof(X)));

   g(p, mystery);

   // possible diagnostic of potential use after

   // invalidation (QoI)

   p->f();

   return p; // risky, but no diagnostic expected (QoI)
```

Question 04: what are the operations that are expected to be diagnosed on an object that has been invalidated through `[[invalidate_dereferencing]]`?

Answer: on pointers, the intent is to react to indirections on invalidated pointers, so uses of `operator*`, `operator->` and `operator[]` are intended to lead to diagnostics.

Question 05: is `[[invalidate_dereferencing]]` intended to work on non-pointer arguments?

Answer: yes, as it allows for support of smart pointers or other pointer-like handles.

Question 06: is `[[invalidate_dereferencing]]` intended to work on references?

Answer: see question 05 in general, but consider the following example usage:

```
// "borrow" a T object from a set of pre-allocated ones, as
// long as that object satisfies predicate "pred"
template <class T, class Pred>
   T& borrow(Pred pred) {
      // ...
   }
// "let go" of an object, promising not to use it anymore. The
// object can be reused elsewhere in the program after this
// function concludes execution
template <class T>
   void let_go(T &obj) {
      // ...
   }
// ...
// client code...
// ...take any X object from the pool (pred is a tautology)
X & x = borrow<X>([](auto &&){ return true; });
// ...
let_go(x);
// it is incorrect to use "x" starting here
```

In this case, we eventually reach a point past which a referred-to object cannot be used anymore. To inform the compiler of this fact, we could write:

```
template <class T, class Pred> T& borrow(Pred pred) {
   // ...
}
template <class T>
   void let_go([[invalidate_dereferencing]] T &obj) {
      // ...
   }
struct X { void f(); /* ... */ };
void test() {
   T &r = borrow<X>([]{ return true; });
   // ... can use r here...
   let_go(r);
   // r.f(); // diagnostic expected here
}
```

Question 07: is there an opt-out?

Answer: no specific opt-out is being proposed for this attribute, but `std::launder()` could play that role:

```
void *allocate(std::size_t);
void *reallocate([[invalidate_dereferencing]] void*, std::size_t);
void *deallocate([[invalidate_dereferencing]] void*);
void f(int)l
// ...
auto p = static_cast<int*>(allocate(100 * sizeof(int)));
// ...
auto q = static_cast<int*>(reallocate(p, 200*sizeof(int)));
if(q == p) { // fine, no dereferencing
   f(*q); // Ok, supposing the lifetime of q[0] has begun
   // f(*p); // diagnostic expected
   f(*std::launder<int*>(p)); // Ok, I guess
}
```