P3442R2 – [[invalidate_dereferencing]] attribute

Document number: P3442R2

Date: 2025-05-18

Authors:

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

Reply to: patricer@gmail.com

Target audience: EWG, SG23, SG14

Revision history

R1 to R2: SG23 has seen R1 and took the following poll: “SG23 supports P3442R1 and forwards it to EWG for further consideration.”

Favor: 8	Neutral: 0	Against: 0

There has been a request for a more thorough discussion of out-of-storage pointers with respect to [basic.compound] p4. On this topic, see On out-of-storage pointers.

There has been a request for a clearer explanation of “invalidate an object” in terms of the core language. On this topic, see On the meaning of “invalidating an object”.

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 related work for details.

Introduction

Some functions take arguments that offer indirect access to objects and invalidate that object for client code usage 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
// diagnosing this is QoI. Standard libraries and compilers
// sometimes 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

In essence, 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. This includes returning the object from a function after `[[invalidate_dereferencing]]` has been applied to it.

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(); /* ... */ };
X *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
    return q; // diagnostic expected here
}
```

Note: for the purposes of this attribute, passing a pointer by value to a function after `[[invalidate_dereferencing]]` has occurred is also expected to be diagnosed:

```
void f(int*);
void g([[invalidate_dereferencing]] int*);
void h() {
  int *p = new int{ 3 };
  f(p); // fine
  g(p); // Ok
  int *q = p; // looking for trouble
  f(p); // diagnostic expected
  delete p; // diagnostic expected
  f(q); // might be diagnosed as QoI (not part of this proposal)
}
```

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 related 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, although that name might be too restrictive.

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.

On the meaning of "invalidating an object"

An interesting question that was raised following R1 of this paper was "*And, beyond raw pointers, what does "invalidate that object" actually mean, in terms of the core language?*".

The original idea behind `[[invalidate_dereferencing]]` was to convey programmer intent to compilers through the source code that would otherwise be opaque, and use that information to produce error messages that would prevent dangerous actions to occur:

```
template <class T>
  T* something_unknown([[invalidate_dereferencing]] T*);
// ...
int f(int n) {
```

```
  int *source(int);
  int *p = source(n);
  // something_unknown(p)
  // return *p // intent: error at this point
  p = something_unknown(p);
  return *p; // intent: Ok, p has been reassigned-to
}
```

Even though that was not the original idea, an avenue suggested by this question is that dereferencing a pointer after it has been passed as an argument annotated with `[[invalidate_dereferencing]]` could be a case of erroneous behavior.

There are indeed similarities: [defns.erroneous] describes erroneous behavior as "well-defined behavior that the implementation is recommended to diagnose", and as such a call to a function with arguments annotated with `[[invalidate_dereferencing]]` could be said to behave *as-if* the argument was assigned the evaluation of an `std::erroneous()` function that yields an erroneous value before the return value of the function is used in the caller.

Thus, the above example:

```
template <class T>
  T* something_unknown([[invalidate_dereferencing]] T*);
int f(int n) {
  int *source(int);
  int *p = source(n);
  something_unknown(p)
  return *p // intent: error at this point
}
```

…would be equivalent to (italicized text is *as-if*):

```
template <class T>
  T* something_unknown([[invalidate_dereferencing]] T*);
int f(int n) {
  int *source(int);
  int *p = source(n);
  something_unknown(p)
  p = std::erroneous();
```

```
  return *p // erroneous read (diagnosis encouraged)
}
```

For a more complete example, the following excerpt:

```
template <class T>
  T* something_unknown([[invalidate_dereferencing]] T*);
int f(int n) {
  int *source(int);
  int *p = source(n);
  p = something_unknown(p);
  return *p; // intent: Ok, p has been reassigned-to
}
```

… would become equivalent to (again, italicized text is *as-if*):

```
template <class T>
  T* something_unknown([[invalidate_dereferencing]] T*);
int f(int n) {
  int *source(int);
  int *p = source(n);
  auto p' = something_unknown(p); // note: notional p' only
  p = std::erroneous();
  p = p';
  return *p; // intent: Ok, p has been reassigned-to
}
```

Of course, an implementation would be free to remove the notional `p'` object and the intermediate assignment of an erroneous value to `p` in this case. Implementations would not be required to write to the "invalidated" object with `std::erroneous()`; the requirement would be to behave as-if that write had occurred.

Since the question was concerned with "beyond raw pointers", we will examine two other use cases: applying `[[invalidate_dereferencing]]` to references and applying it to (non-standard[1]) smart pointers.

[1] For standard smart pointers, P3465 already offers a general approach.

Although we did not go into as much detail for these two cases as for raw pointers in previous versions of this paper[2], the issue of the meaning of "invalidated" could be covered by the same approach, making it erroneous to read from an object passed as an argument to a function when that argument has been annotated with `[[invalidate_dereferencing]]` until that object has been reassigned-to.

For references, take the following example:

```
// "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
```

This example shows a case where the programmer's intent with respect to `let_go()` is inscribed in comments, making it difficult for a compiler to diagnose inappropriate uses of `x` in client code (reads from the `x` object after a call to `let_go()`).

To inform the compiler of the intent, we could write:

```
template <class T, class Pred> T& borrow(Pred pred) {
```

[2] In previous revisions of this paper, they were mentioned in question 05 and question 06 of the FAQ section.

```
   // ...
}
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
}
```

There does not need to be an actual "invalidation" of the object passed to `let_go()` for the `[[invalidate_dereferencing]]` here: indeed, we do not want such a write to be introduced, as the referred-to object might be valid for other uses, but not for the caller of `let_go()`. Obviously, writing to `r` after calling `let_go()` is a bad idea in this case, but not one we are trying to solve as of this moment.

The (non-standard) smart pointer case is similar. Consider the following (simplistic) adaptation of the previous example:

```
template <class T>
  class borrower_ptr {
    T *p;
  public:
    borrower_ptr(T *p) : p { p } {
    }
    T* operator->() const { return p; }
    T& operator*() const { return *p; }
    // etc.
  };
template <class T, class Pred>
```

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

The semantics of `borrower_ptr<T>` are deliberately left vague: maybe `let_go()` resets its `p` data member to `nullptr`, maybe it sets `p` to `std::erroneous()`, maybe it does nothing at all and just serves as a marker in user code stating "I let go of what I borrowed and swear I will not use it from that point on unless I assign to it first".

In this case, again, `[[invalidate_dereferencing]]` would carry the programmer's intent and allow the compiler to identify any use of `p` as being erroneous unless `p` has first been assigned-to.

On out-of-storage pointers

The following questions have been raised with respect to out-of-storage pointers: "*Please represent the core language situation for out-of-storage pointers correctly, per [basic.compound] p4. It's not just dereferencing that's undefined behavior, it's deallocation, too. Also, even an lvalue-to-rvalue conversion on a pointer variable is implementation-defined (and thus might have surprising effects) when the pointer is out-of-storage.*"

It is possible that the questions raised here are due to the current name for this feature. One problem with `[[invalidate_dereferencing]]` is that it conveys well the intent behind the basic idea of "do not use what this pointer points to before reassigning to it" that drove this paper from the onset but might be misleading for the other use cases it targets. For example, the following is expected to be diagnosed even though we are passing `p` to a (deallocation) function and not reading from `p`:

```
void f([[invalidate_dereferencing]] void *);
// ...
int g(int *p) {
  assert(p);
  int n = *p;
  f(p); // use of p past this point is erroneous unless
        // p has been reassigned-to
  free(p); // erroneous use of p (diagnostic expected)
}
```

The question was raised in the context of [basic.compound] p4 which states "*A pointer value P is valid in the context of an evaluation E if P is a pointer to function or a null pointer value, or if it is a pointer to or past the end of an object O and E happens before the end of the duration of the region of storage for O. If a pointer value P is used in an evaluation E and P is not valid in the context of E, then the behavior is undefined if E is an indirection ([expr.unary.op]) or an invocation of a deallocation function ([basic.stc.dynamic.deallocation]), and implementation-defined otherwise.*". This paragraph defines the rules for a pointer to be valid and expresses invalid accesses to lead to undefined behavior, so the question that was raised is indeed understandable.

This is why it's important to go back to the intent, and to remind ourselves that the attribute name is imperfect but serves for the moment as a "working title". Although there are similarities between pointer validity in the context of [basic.compound] p4 and the proposed `[[invalidate_dereferencing]]` attribute, the former describes core language rules for validity of evaluations involving pointers whereas the latter describes an in-source-code marker that informs the compiler that "even though it's not self-evident from context, using the indirectly-referred entity of this function argument past this point is a bad idea and should be diagnosed as such".

Expressed succinctly: the attribute is meant for pointers but not limited to pointers, and there is no requirement to invalidate pointer usage in the sense expressed in [basic.compound] p4, just a request to diagnose such usage as invalid from the "apparent point of invalidation" on. As mentioned previously, that might mean "act as if accessing the pointed object or the referred object from calling code past this point is erroneous behavior".

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. See On the meaning of "invalidating an object" for an example.

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

Answer: yes. See On the meaning of "invalidating an object" for an example.

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
}
```

This might be useful in situations where the object is made valid again through more obscure means:

```
struct X { /* ... */ };
X * obtain(); // obtains a pointer to some X object
void release([[invalidate_dereferencing]] X*); // lets it go
void remap(X **px); // points *px to some valid X object
int main() {
   auto p = obtain();
   // use p
   release(p); // Ok, do not dereference past this point
```

```
    // ...
    remap(&x); // Ok, p might be valid but unclear
    // ...
    // X x = *p; // expected to be diagnosed
    X x = *std::launder(p); // Ok (we trust you)
}
```