Doc. No. P3446R0
Date: 2024-10-14
Audience: SG23, EWG, LEWG
Reply to: Bjarne Stroustrup (bjarne@stroustrup.com)

Profile invalidation – eliminating dangling pointers

Abstract

The invalidation profile is essential for memory safety, requires (relatively simple) static analysis to implement (probably in the compiler), and is probably the most often misunderstood profile. Together with **initialization** and **pointers**, this is the profile that eliminates access through dangling pointers.

This note outlines a design that still needs to be implemented and tested. However, some of the techniques were tried out in the Microsoft Visual Studio implementation of the C++ Core Guidelines. This is a note for comments. It contains some rationale rather than just specification.

Together with **concurrency**, I consider **invalidation** the hardest to specify and the one that requires the most static analysis. Simply following first principles doesn't give an affordable design.

0. Restatement of principles

- Provide complete guarantees that are simple to state; statically enforced where possible and at run-time if not.
- Don't try to validate every correct program. That is impossible and unaffordable; instead reject hard-to-analyze code as overly complex.
- Wherever possible, make the default for common code safe by making simplifying assumptions and verifying them.
- Require annotations only where necessary to simplify analysis. Annotations are distracting, add verbosity, and some can be wrong (introducing the kind of errors they are assumed to help eliminate).
- Wherever possible, verify annotations.
- Do not require annotations on common and usually safe code.
- Do not rely on non-local static analysis.

1. Brief summary

From P3274R0 §3.9

Profile: Invalidation

- ***Definition***: No access through an invalidated pointer or iterator
- ***Initial version***: The compiler bans calls of non-**const** functions on a container when a pointer to an element of the container has been taken. Needs a **[[non-invalidating]]** attribute to avoid massive false positives. For the initial version, allow only straight-line code involving calls of functions on a container that may invalidate a pointer to one of its elements (P2687r0).
- ***Observation***: In its full generality, this requires serious static analysis involving both type analysis and flow analysis. Note that "pointer" here means anything that refers to an object and "container" refers to anything that can hold a value (P2687r0). In this context, a **jthread** is a container.

The definition is clear enough, but we need many details to guide implementation.

I hope the need is obvious. Without something like this, we couldn't claim C++ memory safe however much good other profiles (e.g. initialization and ranges) do.

invalidation is not conceptually difficult and primarily requires enforcement of long-standing rules of C and C++. That doesn't imply that it's easy to implement in exiting tool chains, just that it is relatively simple to see what use cases much be addressed (that's deductive, rather than inventive).

Below ,"pointer" refers to any object that gives direct access to another (§5).

1. Pointer validity

I call a pointer that points to an object or is the **nullptr** a "valid pointer." Pointers are initialized to point to something or the **nullptr** (See **initialization**), i.e., they start out valid. Therefore, when we enter a scope (e.g., a function), we can assume that all pointers from outside the scope are valid. We must keep them valid ("not to dangle") within the scope and when we leave the scope all pointers "exported" to an outer scope must remain valid.

1.1. What is "a pointer"?

In this context, "a pointer" refers to any object that gives direct access to another. The invalidation rules must apply to all such. For this Profile to be comprehensive, we must define "pointer" to include any object that directly or indirectly could contain a pointer:

- Classes with pointer members

- Lambdas (they are classes, and remember capture-by-reference)
- **Jthread**s (they are classes in a defined scope)
- **unique_ptr**s and **shared_ptr**s (they are classes with a single logical pointer member)
- Pointers to pointers
- References to pointers
- Arrays of pointers

Curiously, a pointer-to-member is not a pointer. Rather it is a typed offset to a pointer to an object.

1.2. Deleted pointers

A free-store object that is constructed and destroyed in the same scope shouldn't exist, but thanks to overuse of **new** (rather than local objects and resource-management pointers) such cases are not uncommon. Consider:

```
void f(int x)
{
        int* p = new int{7};
        int* q = new int{8};
        delete q;
        *q = 9;          // likely disaster (disallow)
        If (x) *q = 11;  // possible disaster (disallow)
        if (x) delete p;
        *p = 10;         // likely disaster (disallow)
        If (0<x) *p = 1; // possible disaster (disallow)
}
```

There are four combinations for a static analyzer:

- A pointed-to object is unconditionally deleted and then used. That's easily caught.
- A pointed-to object is unconditionally deleted and then conditionally used. In general, that's impossible to catch statically.
- A pointed-to object is conditionally deleted and then used. In general, that's impossible to catch statically.
- A pointed-to object is conditionally deleted and then conditionally used. In general, that's impossible to catch statically.

We have two choices:

- Catch the deletion of a local pointer
- Catch the use of a deleted pointer

The former is easy to implement and doesn't require flow analysis. Given that strictly local use of free store should be discouraged, I recommend that first (more restrictive and simpler) solution.

In general, we should avoid solutions that are so clever that some implementations will have serious problems enforcing them. In this case, it is easy to construct examples that are impossible to catch statically so accepting them would require run-time checks which are best avoided for performance reasons.

Obviously, **delete p** invalidates every dereference of **p**. Static analysis cannot unaided detect invalidation from other functions. Consider:

```
void use(int* p)            // p is valid because it must be valid in the calling scope
{
        // ...
        delete p;       // possible disaster (disallow)
}

void f(int*p)
{
        use(p);
        *p = 7;         // disaster
}
```

We could address that by using a **shared_ptr** or a **unique_ptr**, but in existing code bases that can be quite intrusive and resource-management pointers will not work through a C-style interface, e.g., the interface to a C function.

The suggested solution is

- Disallow deletion of a “plain” pointer
- Introduce a way of saying that the obligation to **delete** is passed on

This implies that the **delete p** in the example above is disallowed. That’s usually OK because such examples where an object is created in one function and deleted in another should be rare outside constructor/destructor pairs (see §4).

1.3. Escaping pointers

A pointer to an object shouldn’t escape the scope of the object it points to and if it does, it mustn’t be used for access. Consider:

```
void f()
{
        int* p = nullptr;
        int* q = nullptr;
        {
                int x = 7;
                p = &x;
                // ...
                if (x) q = &x;
        }
```

```
        *p = 9;  // likely disaster (disallow)
        *q = 7;  // likely disaster (disallow)

}
```

Local static analysis can catch many such problems. However, the problem is (again) that code introducing the problem (here letting a pointer to a local escape) can be arbitrarily complex.

The simple and obvious solution is to disallow every assignment of a pointer to a local to a non-local pointer, rather than trying to determine which conditional assignment leads to problems.

This solution should be applied to every escaping pointer. For example:

```
int* g()
{
        int x = 7;
        int y = 9;
        // ...
        if (x) return &x;       // likely disaster (disallow)
        return &y;              // likely disaster (disallow)
}
```

Note that pointers to local variables can be safely passed as function arguments. For example, **sort(&v[0],&v[v.size()])**. Given **invalidation**, the called function cannot save a copy of such a pointer for use after its return.

1.4. Pointers to elements

Letting a pointer (usually a reference) escape the scope of the object it points to is common for container access operations. Those are handled by not letting such pointers outlive the object they point into (§1.5).

1.5. Pointers that are meant to “escape”

Returning a pointer to a free-store object or to a static object is not uncommon and safe. However, returning a “raw” (built-in) pointer to a free-store object causes a memory-management problem (the possibility of leaks) for the receiving scope and should be discouraged (§1.2, §3). Pointers that can be passed to the scope of a caller are:

- A pointer that enters from an outside scope (e.g., as an argument)
- The result of **new**
- The pointer to a **static**

. Consider:

```
int* glob = nullptr;

int* f(int* p)
{
        return p;               // OK
        return glob;            // OK
        return new int{7};      // OK (but that has different problems)
}
```

Unless the **int** created by that **new** is meant to live forever we have a memory leak which is caught (§3).

1.6. Pointer escape is viral

Consider

```
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)
}
```

This is a very common style of code: pass a pointer to an object as an argument to a function and then return a pointer to that object back again. For example, that is the style of many C and C++ standard library functions, such as **find()** and **strcpy()**, as well as smart pointers and views.

As a default, we make the assumption that the pointer returned has the same lifetime as the argument. That catches the example above and equivalents. It also allows, the second most common use, returning a pointer that originated in a surrounding scope. For example:

```
std::span<int> foo(std::vector<int>& v)
{
        auto span = make_span(v);
        return span;            // escaping pointer (allow)
}
```

That follows from the rule in §1.5.

This leaves the rare(?) case of a function taking an argument pointer an object of one scope and returning a pointer to an object in an enclosing scope. For example:

```
std::vector<int> glob{1,2,3,4,5};

std::span<int> foo(std::vector<int>& vv)
{
        auto span = make_span(glob);
        return span;            // escaping pointer (allow)
}
```

The reasonable default disallows this last example. To accept this class of examples requires an annotation:

```
std::span<int> foo(std::vector<int>& vv) [[not_local]]
{
        auto span = make_span(glob);
        return span;            // escaping pointer (allow)
}
```

The **span** itself is copied, so that is not a violation.

Note that the annotation is not needed unless a function takes a pointer argument:

```
int* glob()
{
        static int x = 7;
        return x;
}
```

The rules in §1.5 make sure that only pointers to non-local objects can be returned when there aren't any arguments.

1.7. Multiple pointer arguments

Functions that takes more than one pointer and return (one or more) pointers are very common. For example:

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

int check(int* p)
{
        int y = 9;
        return min(&x,&y);      // disallowed
}
```

Because **y** is local we cannot return **min(&x,&y)** (despite – in this simple case – we can see that the valid pointer to **x** is returned). To make analysis affordable, we must accept some false positives.

1.8. Variables and members

An object that is given a pointer to hold must be considered a pointer to that object. "To hold" means that is has a member of pointer type and has had a pointer of that type passed as an argument to a constructor or member function.

Consider this real-world tricky example:

```
class Window {};        // a gui window

class Widget {          // a widget inside a window
        Window* window_;
public:
        Widget(Window& parent_window)  :window_(&parent_window) {}
};

struct Logger {
        void log(char const*) { /* ... */ }
};

class WidgetFactory {           // optionally logs each creation
        Window* window_;
        Logger* logger_;
public:
        WidgetFactory(Window& w, Logger& l) :window_(&w), logger_(&l) {}        // log
        WidgetFactory(Window& w) :window_(&w), logger_(nullptr) {}      // don't log

        Widget create_widget()
        {
                if (logger_)
                        logger_->log("Creating new Widget");
                return Widget{*window_};     // returns object containing a pointer
        }
};

Widget inner(Window& window)
{
        Logger l;
        WidgetFactory factory{window, l};           // factory depends on window and l
        return = factory.create_widget();      // the widget doesn't depend on the logger
}

void outer()
```

```
{
        Window window;
        Widget wid = inner(window);
}
```

Sorry for the length of this example, but real-world and short don't always mesh. I have abbreviated it as much as I could without missing the point.

The point is that in **inner()**, **WidgetFactory** takes two arguments only one of which affects the **Widget** returned. However, without looking into **create_widget()**, we can't know that, so without non-local static analysis we must disallow that example.

We have five choices for this potentially large class of programs:

- Rely on non-local analysis
- Disallow such examples as too complex for offering the invalidation guarantee
- Allow the potential problem to slip through
- Invent some annotation to help the programmer
- Suppress **invalidation** for such code

None are particularly attractive. I see an annotation as the least bad

```
Widget create_widget() [[not_returned(logger)]]
{
        if (logger_)
                logger_->log("Creating new Widget");
        return Widget{*window_};      // returns object containing a pointer
}
```

This ensures that the default is safe (if overly restrictive) and this annotation has the great advantage that it can be verified in most cases (remember the implementation code can be arbitrarily complex). Note that the dependency on **window** (and its lifetime) must be preserved to avoid dangling pointers. That means that a simple, non-parameterized annotation isn't sufficient.

1.9. Avoiding flow control

In general, even local static flow analysis is impossible and it can easily be unaffordable. In in §1.2, I suggest that we – at least initially – treat all declarations as unconditional. That gives false positives but leaves the option open to rewrite declarations so that they are not dependent on conditionals.

For example, without serious local flow-analysis, we must deem **logger_** used here:

```
Widget create_widget()
```

```
{
        if (logger_)
                logger_->log("Creating new Widget");
        return Widget{*window_};      // returns object containing a pointer
  }
```

I think it unwise to require such analysis, and for portability it would even be unwise to allow such analysis. It would be unfortunate if correctness varied with the cleverness of implementers.

1.10. Class hierarchies

Consider:

```
struct B {
};

struct D : B {
        int* p;
};

B* f()
{
        int xx;
        D* dp = new D{7};
        return dp;              // pointer to local pointer tries to escape (disallow)
}
```

The usual rules for pointers seem to handle this. As usual, the potential complexity of code (conditionals, loops, etc.) conditional. Will most likely force us to restrict verified code to straight-line code where we can easily spot information-losing implicit conversions.

2. Invalidating operations on containers

In relation to containers, we define "invalidated" as “when an operation on a container may have reallocated or deleted its elements”, thereby making a pointer to an element invalid. For example:

```
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)
        *p = 7;                 // likely disaster (disallow)
}
```

Three things must be the case for a disaster to happen:

- A pointer to an element must exist (an alias).
- The operation must relocate the element pointed to.
- The element pointed to must be accessed after the relocation.

That's rare, but disastrous when it happens, and hard to prevent without serious overhead (e.g., garbage collection) or restrictions. Here, we take the restriction path.

If we are conservative, we can relatively easily detect the creation of an alias. That is, by unconditionally considering even conditionally created aliases potentially used after relocation.

Furthermore, we can avoid invalidation by allowing only non-**const** operations on a container for which an alias has been taken.

So as a first cut, we need to disallow every non-**const** operation on a container from which a pointer to an element has been taken. That handles the example above, but disallows many non-invalidating functions that are not **const**, for example **vector::operator[]()**. We must accept those.

We need a **[[not_invalidating]]** attribute to mark functions that potentially could invalidate, but don't. Considering non-**const** functions invalidating is a good and safe default. Therefore, the annotation should be used only to add to that; that is, we need **[[not_invalidating]]** rather than **[[invalidating]]**. A **[[not_invalidating]]** annotation can (and should) be statically verified by examining the definition of a supposedly **[[not_invalidating]]** function.

2.1. Invalidation is viral

Unfortunately, marking member functions is by itself not sufficient. For example, the alias may be taken in one function and the invalidating operation in another:

```
void f(vector<int>& vi)
{
        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);
        *p = 7;                 // likely disaster (disallow)
}
```

The fundamental problem here stems from two facts

- Very few functions taking a non-**const** container argument performs invalidating operations.

- An invalidating operation is only bothersome if an alias to one of its elements has been retained elsewhere.

This means that if we disallowed all invalidating operations under **invalidation**, we would suffer a massive (crippling) number of false positives (as well as losing key aspects of the STL). However, to disallow invalidating operations only when an alias for an element exists (as we must) requires that information about aliasing be passed along the call chain at compile time. That must be a form of non-local static analysis – something we have good reasons to avoid.

To guarantee memory safety, we must eliminate dangling pointers. We are between a rock and a hard place:

- If we unconditionally disallow invalidating operations under **invalidation**, users would have to suppress **invalidation** so frequently that it would become routine, and it would be hard to verify that such suppression didn't lead to dangling pointers. In particular, this could lead to dangling pointers in future uses of a function containing suppression.
- Precise detection of invalidation would require flow analysis of every function that could lead to a function with an invalidating operation. That's the worst-case scenario for global static analysis. I consider that unrealistic.

So, an affordable perfect solution to the invalidation question is impossible, so we must look for a good-enough sound solution.

Since we can't statically analyze our way to an answer for realistic code, we need some annotation to help us. **Const** helps a lot, but we must distinguish between invalidating and non-invalidating uses of non-**const** operations. Also, by far the most common operations are non-invalidating, so we should not require those to be annotated. By default, **invalidation** disallows invalidating operations, so the default is safe. Consequently, the annotation must be to selectively allow invalidation.

Suppressing **invalidation** for sections of code is possible but is a crude mechanism that is likely buried deep in code.

To get the permission to invalidate out of the code and onto the function declarations where it is more easily spotted and easier to use in analysis. To permit invalidation, we add an **[[invalidating]]** annotation. For example:

```
void f(vector<int>& vi [[invalidating]])
{
        vi[2] =7;           // not const, but OK
        vi.push_back(9);    // may relocate vi's elements (explicitly allowed)
```

```
}
```

This assumes that **vector::operator[]()** is declared **[[non-invalidating]]**.

By default invalidation is disallowed under invalidation. For example:

```
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)
}
```

Placing the annotation on the parameter, rather than on the use, means that we don't have to examine the definition of a function to ensure safety. This enables a compiler enforcing **invalidation** to

- Catch uses of unintentional invalidation looking only at a single function definition in isolation.
- Catch calls to invalidating functions from functions where an alias has been taken at the point of call.

That is local static analysis.

2.1. Notation

The **[[invalidating]]** on a function argument is ugly, verbose, and potentially confusing. For example:

```
void f(vector<int>& vi [[invalidating]]), vector<double>& vd [[invalidating]]);
```

I see two alternatives:

- Place **[[invalidating]]** on the function declaration itself:

  ```
  void f(vector<int>& vi, vector<double>& vd) [[invalidating]];
  ```

  That's crude and when there are multiple arguments, doesn't say which argument might be invalidated.

- Introduce a type notation

  ```
  void f(invalidating(vector<int>&) vi, invalidating(vector<double>&) vd);
  ```

That will give the compiler the same information as an **[[invalidating]]** attribute with less syntactic noise. On the other hand, we may prefer for some Profiles annotations to be "noicy."

The latter is an example of a choice of notation that emerges for most annotations. Another example is **[[uninitialized]]** vs. an **uninitialized** pseudo-value.

2.2. What is "a container"?

For this context, we must define "container" to include any object that directly or indirectly could contain a pointer:

- Classes with pointer members
- Lambdas (they are classes, and remember capture-by-reference)
- **Jthread**s (they are classes in a defined scope)
- **unique_ptr**s and **shared_ptr**s (they are classes with a single logical pointer member)
- Pointers to pointers
- References to pointers
- Arrays of pointers

As usual, "pointer" can be anything that refers to an object (e.g., a **unique_ptr**). A "pointer member" of one of these can lead to an invalidation problem only if

- it is possible to point to an element pointed to by such a "pointer member"
- the value of the pointer member can be changed

For example:

```
int x = 7;
int* p = &x;
int** pp = &p;
cout << **pp;          // 7
p = nullptr;
cout <<  **pp;         // likely disaster (disallow)
```

Lambdas are function objects, so the rules for classes and objects apply. There may be more specific rules needed to take case of lambda bindings.

3. Ownership

Abstractions that manage a set of objects or just a section of memory are common and essential in many C++ programs. Examples are **vector**, **map**, memory pools and stack allocators. Without such, C++ would not be C++. Most have the property that objects and/or memory are allocated and deallocated by different functions (e.g., constructor-

destructor pairs). This implies that the obligation to **delete**/deallocate must be transferred and/or shared.

Obviously, this could be handled by suppressing validation for such code, but that's an awful lot of often complex code. We can do better with a more specialized annotation.

The problem can be address by distinguishing pointers that must be deleted ("owners") from pointers ("ordinary pointers") that must not.

The C++ Core Guidelines introduced the **owner** alias to address that:

```
template<typename T> using owner = T;
```

To the type system an **owner<T>** is just a **T**, but to a human reader and a static analyzer **owner<T>** indicates an obligation to **delete** or transfer ownership. This is easily enforced by a static analyzer, such as a compiler.

So, as stated in §1.5, **delete** of an ordinary (non-owner) pointer is disallowed.

An **owner** point must either be deleted at the end of the pointer's scope or transferred elsewhere:

- A copy of an **owner** into a non-**owner** simply creates an alias.
- A copy of an **owner** into another **owner** transfers ownership (i.e., the obligation to delete) to the target and deletion and use of the source is disallowed.

Note that when an owner is passed as an **owner** argument to a function, the original owner will be invalid (because the called function must **delete**). For example:

```
void use(owner<int*> p)
{
        // ...
        delete p;       // this delete is required
}

void f()
{
        owner<int*> p = new int{7};
        use(p);
        *p = 7;         // disallow: we know that p has been deleted
}
```

If a class member is an **owner**, a constructor must initialize it (like it must **const** members) and the destructor must **delete** it.

I fear that requiring **owner** annotation on pointers initialized by new will create massive false positives in older code that overuse explicit **new** and **delete**. The alternative would be to make ownership implicit for variables initialized with **new**. That is

```
int* p = new int {7};       // means owner<int*> p = new int {7};
```

owner parameters would still have to be explicitly marked, but that's not a common case.

The **owner** type alias is for human readers and static analyzers, rather than the C++ type system. To be useful in C-style interfaces, **owner** must be a type alias rather than a type. If using a type is feasible, something like **unique_ptr** can be used. However, to implement the simplest such resource managers, something like owner is needed to communicate the need to delete to the destructor. For example:

```
class Int_vec {
        owner<int*> elem;
        int sz;
        int_vec(int s) : elem{new int[s]}, sz{s} {}
        ~int_vec() { delete[] elem; }
};
```

The alternative is to suppress **invalidation** altogether in such implementations.

4.0. Aliases

Aliases are the bane of static analysis, and also of traditional ways of debugging. They are also the backbone of most C and C++ programming styles: every time we pass a pointer (or reference) as a function argument, we create an alias.

Consider

```
owner(int*) p = new int{7};
int* q = p;       // create an alias
delete p;
*q = 9;
```

The **owner** annotation prevents deletion of **q** but not of **p**. Clearly, the use of **q** after the deletion of **p** must be avoided, as must more subtle variants. For example:

```
Int* f(int* p) { return p; }

owner(int*) p = new int{7};
int* q = f(p);     // create an alias
delete p;
*q = 9;
```

Many C and C++ standard-library functions provide variants of that; for example, **strcpy()** and **find()** return pointes to their arguments.

Clearly, the precise rule that we want is that we don't **delete** an object for which there exists an alias, and as pointed out in §1.5 that's not practical for arbitrarily complex code.

The classical solution is to avoid "naked **new**" in application code. Instead, have an object (an "RAII object") that manages the lifetime of the resource (in this case the free-store memory). For example:

```
int* f(int* p) { return p; }

auto p = make_unique(7);
int* q = f(p.get());        // create an alias
*q = 9;
```

Now, the deletion of the **int** is postponed until the end of local scope so the problem with **q** cannot happen. The rule against escaping pointers (§1.5) ensures that the alias cannot "escape" to be misused after the exit from the scope.

So, if we must have **owner** (as I suggest that we must), we need to have its use mirror RAII. That is, a **delete** must be at every point of exit from its **owner**'s scope and not before, and in the right order. This is easily verified statically. Essentially, this is a 40+ year old solution.

3.1. Containers can be aliases

If a container holds a pointer to another container, it must be considered an alias for (pointer to) that other container. Thus, an invalidator of "the holder" must be considered an invalidation of that other container.

Consider:

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

void foo()
{
        std::vector<int> v{1,2,3,4,5};
        Holder inv;
        inv.store_vector(v);    // inv becomes an alias for v
        // ...
        inv.clear();      // not const: assumed to invalidate inv (disallow)
```

```
        v[1] =7;        // v has been invalidated
}
```

The rule for multiple pointers applies to containers of pointers also. For example:

```
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) {
        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, it2] = zip(v, v2);       // returns a local and a non-local
        return it1;       // may return a pointer to the local v2 (disallow)
}

void outer()
{
        std::vector<int> v{1,2,3,4,5};
        auto it = inner(v);
}
```

Here, the **zip()** takes two pointers and must be assumed to return a pointer to the local. Note that we cannot, with a reasonable amount of analysis, make assumptions on the actual lifetime of the two returned pointers, so we must assume the shortest.

4. Comment

I consider **invalidation** conceptionally simple, but hard to implement because the varieties of "pointers" and "containers" makes for a huge number of cases for an analyzer (e.g., a compiler) to consider. If I was in a position to implement the profile myself, I would start by implementing the rules and annotations for "plain old pointers" to get a better feel of the magnitude of the task (in a given C++ implementation), to be able to test the rules, and to start getting a feel of how this profile affects larger older code bases.

5. Acknowledgements

Many thanks to Xavier Bonaventura, Ilya Burylov, Christoff Meerwald, Andreas Weiss, Michael for comments and especially tricky examples.

Some of the ideas were tried out in the C++ Core Guidelines.