P1906R0 - Provided operator= return lvalue-ref on rvalue

Peter Sommerlad

2019-10-07

Document Number:	P1906R0
Date:	2019-10-07
Project:	Programming Language C++ Programming Language Vulnerabilities C++
Audience:	EWGI/EWG/CWG and SG12/WG23

1 Introduction

When preparing my CPPCon 2019 talk I tried to explain that the only typical problem inherited from C is "dangling". During his CPPCon 2019 keynote Sean Parent introduced the terms "relationship type" and "relationship object", meaning an object that relates to another object. Changes to or the end of the lifetime of that referred object can sever the relationship and thus invalidate the state of the relationship object. This is what commonly is called "dangling" or "use after free". In my own CPPCon talk I referred to such "relationship types" as "designating types", which I now consider a worse name. Typical examples of relationship types are references, pointers, views, span, and iterators.

1.1 The Problem: Immediate Dangling

A recent trend I observed (and am guilty of as well) in example code shown in presentations, blog posts and social media is to rely on the lifetime extension of temporary objects by binding them to temporaries. This lifetime extension mechanic is needed to be able to pass temporaries down the call chain via references, but is dangerous when applied with local references. Small changes might lead to binding the local reference to a returned reference instead of the materialized temporary leading to immediate dangling. Even worse are non-reference relationship types that are often Regular types that immediate dangle when returned from member function on a temporary object, such as `vector::data()`.

C++11 introduced reference qualifiers for member functions that can prohibit calling member functions on temporaries (rvalues), especially those that return a relationship object referring to `*this`, a non-static data member of `*this`, or data managed by `*this`, such as elements in a vector. Non-member versions of corresponding functionality, such as `std::begin()` do not have the same problem, because for them the special rule for unqualified member functions [over.maptch.funcs] p5.1

does not apply. Views from std::ranges also do never bind to temporaries, because the overloads taking an rvalue-reference are deleted.

The standard library suffers from that legacy that was never considered or cleaned after ref-qualifiers for member functions were introduced. But that is a topic of another future paper.

This paper is about core language wording. For class types, if nothing is specified the compiler provides copy and move assignment operators as unqualified member functions that return an lvalue reference to the assigned to object ([class.copy.assign] p.2 and p.5). As unqualified members they can be called on temporary objects and rvalues. This violates Scott Meyers' principle "do as the ints do" for operator overloading.

A summary is given in the following table by providing the synthesized signatures for each kind of member functions `mf` for a class type `X`.

	`(X)`	`(X const)`	`(X&)`	`(X&&)`	`(X const &)`	`(X const &&)`
`mf()`	+	-	+	+	-	-
`mfc() const`	+	+	+	+	+	+
`mflc() const &`	+	+	+	+	+	+
`mfr() &`	+	-	+	-	-	-
`mfr() const &`	-	+	-	-	+	-
`mfr() &&`	-	-	-	+	-	-
`mfr() const &&`	-	-	-	-	-	+

Table 1: member bindings with respect to reference categories (full set see also https://godbolt.org/z/kgSQrm)

[*Example*:

```
#include <complex>

template <typename INT>
struct check{
        INT x{};
        INT side_effect(INT &lv){
                lv = INT{42};
                return lv;
        }
        void demo(){
                auto &dangle = (INT{} = INT{42});
                auto side = side_effect(dangle);
                side = side_effect(INT{} = INT{42});
        }
};

struct Int {
        int val{};
};

struct SafeInt {
```

```
        int val{};
        constexpr SafeInt& operator=(SafeInt const &other) & = default;
        constexpr SafeInt& operator=(SafeInt &&other) & = default;
        // need to provide copy move ctor and default ctor as well
        constexpr SafeInt() = default;
        constexpr SafeInt(SafeInt const &other) = default;
        constexpr SafeInt(SafeInt &&other) = default;
};


int main() {
        check<Int> c{};
        c.demo(); // compiles
        check<int> fail{};
        fail.demo(); // compile error
        check<std::complex<double>> cc{};
        cc.demo(); // compiles
        check<SafeInt> failstoo{};
        failstoo.demo(); // compile error
}
```

The `demo()` member function calls on variables fail and failstoo do not compile with the following error messages. The other versions happily produce UB.

```
15:52:40 **** Incremental Build of configuration Debug for project p1906_examples ****
make all
Building file: ../src/p1906_examples.cpp
Invoking: GCC C++ Compiler
g++ -std=c++17 -O0 -g3 -pedantic -pedantic-errors -Wall -Wextra -Werror -Wconversion -c -fmessage-leng
../src/p1906_examples.cpp: In instantiation of 'void check<INT>::demo() [with INT = int]':
../src/p1906_examples.cpp:36:12:   required from here
../src/p1906_examples.cpp:11:25: error: using rvalue as lvalue [-fpermissive]
   11 |   auto &dangle = (INT{} = INT{42});
      |                   ~~~~~~~^~~~~~~~~~
../src/p1906_examples.cpp:13:28: error: using rvalue as lvalue [-fpermissive]
   13 |   side = side_effect(INT{} = INT{42});
      |                      ~~~~~~~^~~~~~~~~
../src/p1906_examples.cpp: In instantiation of 'void check<INT>::demo() [with INT = SafeInt]':
../src/p1906_examples.cpp:40:16:   required from here
../src/p1906_examples.cpp:11:25: error: passing 'SafeInt' as 'this' argument discards qualifiers [-fpe
   11 |   auto &dangle = (INT{} = INT{42});
      |                   ~~~~~~~^~~~~~~~~~
../src/p1906_examples.cpp:24:21: note:   in call to 'constexpr SafeInt& SafeInt::operator=(SafeInt&&)
   24 |  constexpr SafeInt& operator=(SafeInt &&other) & = default;
      |                     ^~~~~~~~
../src/p1906_examples.cpp:13:28: error: passing 'SafeInt' as 'this' argument discards qualifiers [-fpe
   13 |   side = side_effect(INT{} = INT{42});
      |                      ~~~~~~~^~~~~~~~~
../src/p1906_examples.cpp:24:21: note:   in call to 'constexpr SafeInt& SafeInt::operator=(SafeInt&&)
   24 |  constexpr SafeInt& operator=(SafeInt &&other) & = default;
      |                     ^~~~~~~~
```

```
make: *** [src/p1906_examples.o] Error 1
"make all" terminated with exit code 2. Build might be incomplete.

15:52:41 Build Failed. 5 errors, 0 warnings. (took 562ms)
```

— *end example*]

Why is this a problem now?

Well it has long been a problem, but today C++ is used more and more for safety critical systems instead of previously used C, e.g., in the automotive sector (AUTOSAR Adaptive C++ programming guidelines). So we can expect that more and more programmers coming from C or other languages come to C++ to build safety critical systems. For familiar built-in types assignment to an rvalue is prohibited, but it might be advisable to use wrapper classes to avoid mistakes like units or dimension mismatch in arithmetic, e.g., by adding apples and pears (this is an application of the Whole Value design pattern from Ward Cunningham's CHECKS pattern language). With such class types, assignment, e.g., as a typo where comparison was meant, can be called with an rvalue on the left-hand side. In addition I expect that C++ novices can become inventive, by casting an rvalue to an lvalue through assignment to pass rvalues indirectly to lvalue-taking functions, such as the view constructors of std::ranges.

1.2 Potential Remedies

While one can argue that static analysis tools can detect and diagnose 'abominational' uses of assignment operators, I believe the assignment operators provided by the compiler should not lie about their return value and as with the built-in types should only work with lvalues on the left-hand side.

There is one option to special case [over.maptch.funcs] p5.1 for compiler-provided non-deleted assignment operators, or as I suggest below, to add non-const lvalue ref qualification to the compiler-provided assignment operators. From some discussion on the 'ext' mailing list, I came up with a further idea to actually see the unqualified members as implicitly providing two overloads. That leads to the second proposal as suggested by Jens Maurer, to actually provide two overloads, each preserving its value category.

Unfortunately, I do not have the means to check the impact of such a small but significant change to the language. However, I believe that usages that rely on assigning to an rvalue of a class type by the compiler provided assignment operators is broken anyway. Help by those with large enough code bases and query tools is highly appreciated and results can be shown in future versions of this paper.

It has further be considered if such a fix can and should be considered a bug in all previous versions of the language.

2 Wording

The suggested changes are relative to N5410 - the C++20 CD ballot document. The corresponding positions in older revisions of the standard are as follows, if this is to be considered a bug since.

2011 section [class.copy] p.18 and p.21

2014 section [class.copy] p.18 and p.21

2017 section [class.copy.assign] p.2 and p.5

I implicitly provide two versions for the change. First one, just adding a lvalue-ref qualifier to the provided assignment operators. The second one, provide each a second overload preserving the rvalue value category by returning `X&&` from an rvalue-ref qualified overload.

1 *Change [class.copy.assign] (11.3.5. Copy/move assignment operator) p.2 as follows:*

2 If the class definition does not explicitly declare a copy assignment operator, one is declared *implicitly.* If the class definition declares a move constructor or move assignment operator, the implicitly declared copy assignment operator is defined as deleted; otherwise, it is defined as defaulted[dcl.fct.def]. The latter case is deprecated if the class has a user-declared copy constructor or a user-declared destructor [depr.impldec]. The implicitly-declared copy assignment operator for a class `X` will have the form

```
X& X::operator=(const X&) &
X&& X::operator=(const X&) && // 2nd suggestion
```

if

(2.1) — each direct base class `B` of `X` has a copy assignment operator whose parameter is of type `const B&`, `const volatile B&`, or `B`, and

(2.2) — for all the non-static data members of `X` that are of a class type `M` (or array thereof), each such class type has a copy assignment operator whose parameter is of type `const M&`, `const volatile M&`, or `M`.[1]

Otherwise, the implicitly-declared copy assignment operator will have the form

```
X& X::operator=(X&) &
X&& X::operator=(X&) && // 2nd suggestion
```

3 *And change p.5 as follows, removing the inconsistent semicolon is an editorial drive-by fix:*

5 The implicitly-declared move assignment operator for a class `X` will have the form

```
X& X::operator=(X&&) &;
X&& X::operator=(X&&) && // 2nd suggestion
```

3 Recommendations for SG12 and WG23

I suggest that recommendations are added to existing guidelines to not embed assignment operations for objects of class type in other expressions. This is already common practice for conditional statements, where assignment might be a misspelled equality comparison.

I further suggest that user-defined classes that need to overload assignment operators do so as ref-qualified versions as the `SafeInt` class from the example.

---

1) This implies that the reference parameter of the implicitly-declared copy assignment operator cannot bind to a `volatile` lvalue; see **??**.

As a corollary, all member functions should be ref-qualified if they are returning a relationship object referring to `*this`, a member of or data entity managed by `*this`.

4 Appendix

Here a short test program to show, which member functions bind to which value category, also available to play with at https://godbolt.org/z/kgSQrm

```
#include <array>
#include <type_traits>
using one = char;
static_assert(1==sizeof(one));
using two = std::array<char,2>;
static_assert(sizeof(two)==2);
using three = std::array<char,3>;
static_assert(sizeof(three)==3);
using four = std::array<char,4>;
static_assert(sizeof(four)==4);

struct Y{
        one m() { return {};}

        one
        mf() { return {};}
        two mf() const { return {};}

        one mfc() const { return {};}

    one mfr() & { return {};}
    two mfr() && { return {};}
    three mfr() const & { return {};}
    four mfr() const && { return {};}

    one mfl() & { return {};}
    two mfl() const & { return {};}

    one mflc() const & { return {};}

    one mfx() && { return {};}
    two mfx() const && { return {};}

    one mfxnc() && { return {};}

};
static_assert(1 == sizeof(std::declval<Y>().m()));
//static_assert(1 == sizeof(std::declval<Y const>().m()));
static_assert(1 == sizeof(std::declval<Y&>().m()));
static_assert(1 == sizeof(std::declval<Y&&>().m()));
//static_assert(1 == sizeof(std::declval<Y const &>().m()));
```

```
//static_assert(1 == sizeof(std::declval<Y const &&>().m()));


static_assert(1 == sizeof(std::declval<Y>().mf()));
static_assert(2 == sizeof(std::declval<Y const>().mf()));
static_assert(1 == sizeof(std::declval<Y&>().mf()));
static_assert(1 == sizeof(std::declval<Y&&>().mf()));
static_assert(2 == sizeof(std::declval<Y const &>().mf()));
static_assert(2 == sizeof(std::declval<Y const &&>().mf()));

static_assert(1 == sizeof(std::declval<Y>().mfc()));
static_assert(1 == sizeof(std::declval<Y const>().mfc()));
static_assert(1 == sizeof(std::declval<Y&>().mfc()));
static_assert(1 == sizeof(std::declval<Y&&>().mfc()));
static_assert(1 == sizeof(std::declval<Y const &>().mfc()));
static_assert(1 == sizeof(std::declval<Y const &&>().mfc()));


static_assert(2 == sizeof(std::declval<Y>().mfr()));
static_assert(4 == sizeof(std::declval<Y const>().mfr()));
static_assert(1 == sizeof(std::declval<Y &>().mfr()));
static_assert(2 == sizeof(std::declval<Y &&>().mfr()));
static_assert(3 == sizeof(std::declval<Y const &>().mfr()));
static_assert(4 == sizeof(std::declval<Y const &&>().mfr()));

static_assert(2 == sizeof(std::declval<Y>().mfl()));
static_assert(2 == sizeof(std::declval<Y const>().mfl()));
static_assert(1 == sizeof(std::declval<Y&>().mfl()));
static_assert(2 == sizeof(std::declval<Y&&>().mfl()));
static_assert(2 == sizeof(std::declval<Y const &>().mfl()));
static_assert(2 == sizeof(std::declval<Y const &&>().mfl()));

static_assert(1 == sizeof(std::declval<Y>().mflc()));
static_assert(1 == sizeof(std::declval<Y const>().mflc()));
static_assert(1 == sizeof(std::declval<Y&>().mflc()));
static_assert(1 == sizeof(std::declval<Y&&>().mflc()));
static_assert(1 == sizeof(std::declval<Y const &>().mflc()));
static_assert(1 == sizeof(std::declval<Y const &&>().mflc()));


static_assert(1 == sizeof(std::declval<Y>().mfx()));
static_assert(2 == sizeof(std::declval<Y const>().mfx()));
//static_assert(1 == sizeof(std::declval<Y&>().mfx()));
static_assert(1 == sizeof(std::declval<Y&&>().mfx()));
//static_assert(2 == sizeof(std::declval<Y const &>().mfx()));
static_assert(2 == sizeof(std::declval<Y const &&>().mfx()));

static_assert(1 == sizeof(std::declval<Y>().mfxnc()));
//static_assert(1 == sizeof(std::declval<Y const>().mfxnc()));
//static_assert(1 == sizeof(std::declval<Y&>().mfxnc()));
```

```
static_assert(1 == sizeof(std::declval<Y&&>().mfxnc()));
//static_assert(1 == sizeof(std::declval<Y const &>().mfxnc()));
//static_assert(1 == sizeof(std::declval<Y const &&>().mfxnc()));
```