

Project: ISO JTC1/SC22/WG21: Programming Language C++
 Doc No: WG21 **P2644R1**
 Date: 2022-11-11
 Reply to: Nicolai Josuttis (nico@josuttis.de)
 Co-authors: Herb Sutter, Titus Winter, Hana Dusíková, Fabio Fracassi, Victor Zverovich, Bryce Adelstein Lelbach, Peter Sommerlad
 Audience: EWG, CWG
 Issues: [cwg900](#), [cwg1498](#), [ewg120](#)
 Previous: <http://wg21.link/P2012>

Final Fix of Broken Range-based `for` Loop, Rev 1

This paper summarizes the fix for the still open issues [cwg900](#), [cwg1498](#), [ewg120](#). The issue is a bug that is 13 years old now, applies to one of the most important control structures of Modern C++, and leads to confusion and ill-formed programs due to unexpected undefined behavior and effort for teaching and training.

We agreed on going the path proposed here with <http://wg21.link/P2012>. But while that paper went in the right direction, some people wanted to have a broader fix.

Because until now, no additional things were proposed for C++23, we fall back to the proposed improvement that was highly agreed on. This fix will not disable future broader fixes.

Tony Table:

	Before	After
<code>for (auto e : getTmp().getRef())</code>	BROKEN	OK
<code>for (auto e : getVector()[0])</code>	BROKEN	OK
<code>for (auto valueElem : getMap()["key"])</code>	BROKEN	OK
<code>for (auto e : get<0>(getTuple()))</code>	BROKEN	OK
<code>for (auto e : getOptionalColl().value())</code>	BROKEN	OK
<code>for (char c : get<string>(getVariant()))</code>	BROKEN	OK
<code>for (auto s : std::span{arrOfConst()}.last(2))</code>	BROKEN	OK
<code>for (auto e : std::span(getVector().data(), 2))</code>	BROKEN	OK
<code>for (auto e: co_await coroReturningRef())</code>	BROKEN	OK
// assume getValue() returns value by reference:		
<code>for (char c : getData().value)</code>	OK	OK
<code>for (char c : getData().getValue())</code>	BROKEN	OK

This means that this paper fixes unexpected and surprising behavior of containers, tuples, optionals, variants, spans, getters that return references, and even coroutines that yield references.

As the last row of the table demonstrates, the proposed solution especially helps not to run into the trap of UB when switching from direct member access to getters.

See <https://www.godbolt.org/z/WPjnx3Mja> for the full example of the broken code.

For the example of the broken code using coroutines, see

https://twitter.com/hankadusikova/status/1542244987882115075?s=20&t=E0y0Prm_RmNeHOZdBuzLVw

Background

This fix was discussed in detail a lot according to <http://wg21.link/P2012>.

See <https://wiki.edg.com/bin/view/Wg21telecons2021/EWG-2021-01-28> which resulted in the following vote:

There is a problem to be solved with range-based for loops and lifetime of temporaries.

SFF	NASA
17	10200

However, when discussing the final wording, some attendees wanted a more broad solution. Therefore, the proposed solution was not accepted yet.

See <https://wiki.edg.com/bin/view/Wg21telecons2021/EWG-2021-09-29>.

So far such alternative was not proposed yet and C++23 is about to be shipped without an improvement of the situation.

Because we have this bug now for 13 years and this affects even beginners, we suggest to finally accept the proposed fix as discussed.

For more details, see <http://wg21.link/P2012>.

Q&A

The three key questions are answered here. For more details, see <http://wg21.link/P2012>.

Do we have evidence that this is a major problem in practice?

We see this problem in practice. Even the authors of this paper ran into this problem. We also know that in all trainings explaining this problem takes significant time.

In addition, more and more style guides warn about using the range-based for loop due to this problem:

- See for example the categorization of the range-based for loop as only “**Conditionally Safe**” in “*Embracing Modern C++ Safely*” by Rostislav Khlebnikov and John Lakos (Bloomberg, 2018).
- <https://abseil.io/tips/107> gives a warning about using the range-based for loop that way (without explaining that the problem is the way the loop is defined).
- The new MISRA standard will constrain using the range-based for loop:
Rule 000389 : A for-range-initializer shall contain at most one function call

Is existing code broken by the fix?

It is possible, but we do not expect that this fix will break existing code. So we did a research.

Here is the result of a check in a very very large code base (Google):

We were able to cobble together a rough analysis: which destructors are invoked on the right hand side of the ":" in a RBF. Running that over a random subset of our codebase, we infer that there are perhaps 10K d'tors in that position. Reducing those and grouping by the relevant types, we can find 0 instances of types in that place that would be a problem. If there were instances that escaped this analysis, we expect that it's on the order of <1 instance per 100MLoC.

But we found something interesting by doing the check: **The current definition of the range-based for loop makes code already unnecessary complex**, because the result continues as follows:

Many (most?) of the d'tors we can find in that location are for utilities that were written specifically to avoid the bug you're proposing to address.

So, it seems the current problem of the range-based `for` loop causes significant drawback in existing code. The person doing the check with the code base summarizes:

Which is to say, for comparison: every deprecation and removal and "nobody will be hurt by this" change that WG21 has made in the past few years (std::random, std::bind1st, changing converting constructor behavior for variant) is 10x+ harder to adopt than this change, as near as we can tell.

How about all the workaround with the initializing range-based `for` loop?

Workaround will not help as long as programmers don't see and understand the problem. However, this problem only sometimes visible (UB) and highly counter-intuitive.

C++ standard should not programmers pay the price for details only experts understand. Especially not in *the* basic control structure. Experts can still have the behavior they want.

Proposed Wording

(All against N4917)

In 6.7.7 Temporary objects [class.temporary]

5 There are ~~three~~ four contexts in which temporaries are destroyed at a different point than the end of the full-expression.

...

7 The fourth context is when a temporary object other than a function parameter object is created in the *for-range-initializer* of a range-based `for` statement. If such a temporary object would otherwise be destroyed at the end of the *for-range-initializer* full-expression, the object persists until the completion of the statement.

???:

Such temporary objects are active everywhere in the scope of the range-based `for` statement and will be destroyed in the reverse order of their construction after the destruction of all variables declared within the range-based `for` statement.

In 8.6.5 The range-based `for` statement [stmt.ranged] add before Example 1:

[*Note*: The lifetime of some temporaries in the *for-range-initializer* is extended to cover the entire loop (6.7.7 [class.temporary]). --- *end note*]

[*Example*:

```
using T = std::list<int>;
const T& f1(const T& t) { return t; }
const T& f2(T t) { return t; }
T g();
void foo() {
    for (auto e : f1(g())) {} // OK, lifetime of return value of g() extended
    for (auto e : f2(g())) {} // undefined behavior
}
```

--- *end example*]

Add a new section in Annex C:

Affected subclause: 8.6.5 [stmt.ranged]

Change: The lifetime of temporary objects in the *for-range-initializer* is extended until the end of the loop (6.7.7 [class.temporary]).

Rationale: Improve usability of the range-based `for` loop.

Effect on original feature: Destructors of some temporary objects are invoked later.

[*Example1*:

```
class NamedData {
```

```
    std::string name;
    std::vector<int> data;
public:
    NamedData(auto n, auto d) : name{n}, data{d} {}
    auto getName() const { return name; }
    const auto& getData() const { return data; }
};

template<typename T>
class LockedAccess {
    T* ptr;
    std::lock_guard<std::mutex> lg;
public:
    LockedAccess(T& o, std::mutex& mx) : ptr{&o}, lg{mx} {}
    auto operator->() { return ptr; }
};

Void foo()
{
    NamedData d{"Test", std::vector{1, 2, 3, 4, 5}};
    std::mutex dMx;
    LockedAccess la{d, dMx};
    for (auto elem : la->getData()) {
        auto n = LockedAccess{d, dMx}->getName(); // now deadlock
    }
}
-- end example]
```

Feature Test Macro

Provide a new value for `__cpp_range_based_for`

Acknowledgements

Thanks to a lot of people who helped and gave support again and again to come to finally get this proposal done.

Rev1:

Updates of wording and examples after discussion in CWG.

Rev0:

First initial version after several versions of <http://wg21.link/P2012>.