P2012R2
Fix the Range-Based `for` Loop

Nicolai M. Josuttis
nico@josuttis.de

09/21

Basic Problems/Symptoms

```
for (auto e : getTmpColl())                          // OK
for (auto e : getTmpColl().getRef())                 // runtime ERROR
for (char c : getVectorOfStrings()[0])               // runtime ERROR
for (auto e : getOptionalVector().value())           // runtime ERROR
for (auto e : std::get<0>(getTuple()))               // runtime ERROR
for (auto e : std::span{getColl().data(), 5})        // runtime ERROR
```

or `std::views::counted(...)`

- Core dump at best
- Some compilers detect this problem **but** only for standard types

http://wg21.link/cwg900:
... the only place where binding a reference to a temporary extends its lifetime implicitly, **unseen** by the user.

Style Guides Warn About / Disable the Range-Based `for` Loop

- **Embracing Modern C++ *Safely***
  by Rostislav Khlebnikov and John Lakos
  Revised March 29th, 2018
  - **"Conditionally Safe Features:"**
    - "Finally, range-based **`for`** loops might hide issues with iterator invalidation and reference lifetime extension, leading to undefined behavior"

- **https://abseil.io/tips/107**

```
// Lifetime extension *doesn't work* here: sub_protos (a repeated field)
// is destroyed by MyProto going out of scope, and the lifetime extension rules
// don't kick in here to magically lifetime extend the MyProto returned by
// GetProto().  The sub-object lifetime extension only works for simple
// is-a-member-of relationships: the compiler doesn't see that sub_protos()
// itself returning a reference to an sub-object of the outer temporary.
for (const SubProto& p : GetProto().sub_protos()) { }   // WRONG
```

- **Draft MISRA C++ coding standards for safety-critical systems:**
  - "A *for-range-initializer* shall contain at most one function call"

3

History

- **http://wg21.link/cwg900 and http://wg21.link/cwg1498 raised exactly this problem in 2009 and 2012 which then 2014 became http://wg21.link/ewg120**

- **EWG notes from Rapperswil 2014:**
  - EWG wants a solution, and welcomes a paper tackling the issue. Vandevoorde raised concerns introducing any new lifetime models. Stroustrup pointed out that the end-of-full-expression rule came about to reduce memory footprint compared to the end-of-block rule and is good for RAII uses. Is it possible to solve the issue by just modifying the specification of a range-for loop?

- **CWG notes from the February, 2017 meeting:**
  - CWG felt were inclined to accept the suggested change but felt that EWG involvement was necessary prior to such a decision.

- **Proposed solution covers all concerns raised**

4

Status (from EWG 2021-01-28)

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

SF	F	N	A	SA
17	10	2	0	0

A solution which might break existing code (such as the lock example Nico showed) is acceptable.

SF	F	N	A	SA
3	15	7	4	0

Against: I would prefer to avoid breaking code, especially since it's not a local effect and can't be detected on compile time.

A solution which proposes a new kind of loop is worth exploring

SF	F	N	A	SA
1	6	10	8	3

Favor: I would like an additional syntax for "safe" for loop

5

Other Safe `for` Loop?

- *This is a problem programmers have when using the range-based for loop as is.*
- *To avoid the problem:*
  - *Programmers have to be aware of the problem*
  - *Know that there is something better (easier/safer) to use*
- We would have to
  - Define yet another loop
  - Teach 4 Millions programmers why it is better
    - Note that we do not want to deprecate the existing loop
  - But still have the problem
    - Note: This is ***the*** loop to iterate over collections
- **Could compilers detect broken code?**
  - Compilers could warn if the right hand side of the range-based `for` loop calls a function returning a reference to a temporary object
  - AND the destructor of the temporary object is not empty
  - Similar to the way we detect lifetime extension problems right now

  We do not expect many false positives.

6

How much code is broken in practice?

Titus checked with a person at Google code base and they reported:

- *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 research:
  The current definition of the range-based `for` loop makes code already unnecessary complex:
  - *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.
- *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.*

7

Proposed Wording for C++23 1/2

Rejected 2021-09-29
(not enough consensus)

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 is created in the *for-range-initializer* of a range-based `for` statement. Such a temporary object persists until the completion of the statement.

In [stmt.ranged] add before Example 1:

[Note: The lifetime of temporaries that would be destroyed at the end of the full-expression of the /for-range-initializer/ is extended to cover the entire loop (class.temporary).]

Add a new section in Annex C:

...

Thanks to Barry Revzin and Jens Maurer for this wording

8

Proposed Wording for C++23 2/2

Rejected 2021-09-29
(not enough consensus)

Add a new section in Annex C:

Affected subclause: 6.7.7 [class.temporary]

Change: The lifetime of all temporary objects in the *for-range-initializer* persists until the end of the loop.

Rationale: Because when the range-base-initializer is an expression that yields a reference to a temporary object created there, the loop iterates over destroyed elements.

Effect on original feature: Valid C++ 2020 code may have different semantics in this revision of C++.

[Example1:

```
std::mutex m;
std::vector<int> v;
// ...
for (auto e: (std::scoped_lock{m}, v)) {
      // m is held across the entire loop; previously m was released before executing the loop body
   // ...
}
```

-- end example]

Thanks to Barry Revzin and Jens Maurer for this wording

9