

Project: ISO JTC1/SC22/WG21: Programming Language C++
 Doc No: WG21 **P2012R2**
 Date: 2020-09-29
 Reply to: Nicolai Josuttis (nico@josuttis.de), Victor Zverovich (victor.zverovich@gmail.com),
 Filipe Mulonde (filipemulonde@gmail.com), Arthur O'Dwyer (arthur.j.odwyer@gmail.com)
 Audience: EWG, CWG
 Issues: [cwg900](#), [cwg1498](#), [ewg120](#)

Fix the range-based for loop, Rev 2

This paper was requested by EWG in 2014 when discussing EWG issue 120 (<http://wg21.link/ewg120>) as a fix for CWG issue 900 (<http://wg21.link/cwg900>).

The range-based for loop became the most important control structure of modern C++. It is **the** loop to deal with all elements of a container/collection/range.

However, due to the way it is currently defined, it can easily introduce lifetime problems in non-trivial but simple applications implemented by ordinary application programmers. This

- is a significant risk in safety-critical contexts
- makes teaching the range-based for loop a problem

We have to

- **Hope** that programmers don't fall into the trap
- Teach beginners significant **constraints and security risks of using** the loop
- Teach more experienced programmers about the **details** of the problem:
 - Explain how the loop is defined
 - Explain lifetime rules for references
 - Explain auto&&
- Teach programmers how to instrument compilers to detect that problem in their code
- Teach programmers about the alternatives (such as using the range-based for loop with initialization) and to mark the alternatives so that they are not accidentally "fixed" introducing the bug again

The far better option is to fix the range-based for loop.

This is what this papers proposes.

Tony Table:

Before	After
for (auto e : getTmp().getRef()) // UB if getRef() returns reference	for (auto e : getTmp().getRef()) // always OK
auto tmp = getTemp(); for (auto e : tmp.getMemByRef())	for (auto e : getTmp().getMemByRef())
// assume we know that we have elements: for (auto e : getVector()[0]) { // UB	// assume we know that we have elements: for (auto e : getVector()[0]) { // OK
for (auto e : get<0>(getTuple())) // UB	for (auto e : get<0>(getTuple())) // OK
for (int i : getOptionalInts().value()) // UB	for (int i : getOptionalInts().value()) // OK
for (int i : std::span(getVec().data(), 5)) // UB	for (int i : std::span(getVec().data(), 5)) // OK
for (auto e : getColl() transform(..) filter(..) // UB if would support rvalues	for (auto e : getColl() transform(..) filter(..) // OK if would support rvalues
const auto& tmp = getTmp().getMemByRef(); // UB when using tmp	const auto& tmp = getTmp().getMemByRef(); // still UB when using tmp (no change)

Rev 2:

Improved proposed wording and minor fixes.

Rev1:

Updates: Discussing whether code can be broken and that the probability of this is close to zero (based on concrete research in real code bases). Discussing why workarounds (such as a new loop) are not a solution. Discussing how compilers could diagnose about broken code.

Rev0:

First initial version.

Motivation

The symptom

Consider the following code examples when iterating over elements of an element of a collection:

```
std::vector<std::string> createStrings(); // forward declaration
...
for (std::string s : createStrings()) ... // OK

for (char c : createStrings().at(0)) ... // UB (fatal runtime error)
```

While iterating over a temporary return value works fine, iterating over a **reference to a temporary return value** is undefined behavior.

Therefor also:

```
// assume we know that createStrings() is never empty here:
for (char c : createStrings()[0]) ... // UB (fatal runtime error)
for (char c : createStrings().front()) ... // UB (fatal runtime error)
```

For the same reason, iterating over the elements of a returned optional collection, is a runtime error:

```
std::optional<std::vector<int>> createOptInts();
...
for (int i : createOptInts().value()) ... // UB (fatal runtime error)
```

This does not only apply to standard types. When iterating over elements returned by a getter we run into the same problem (yes, if the getters returns by value it would work):

```
class Person {
private:
    std::vector<int> values{1, 2, 3, 4};
public:
    const auto& getValues() const {
        return values;
    }
};

Person createPerson();

for (auto elem : createPerson().getValues()) { // UB (fatal runtime error)
    std::cout << "value: " << elem << "\n";
    break;
}
```

See <https://wandbox.org/permlink/ohuuTOyx5k8MWWyh> for demonstrating the last problem in a full example. Depending on the compiler and the platform used, the loop might:

- Print "value: 1" (assuming the value is still there)
- Print "value: 0" (printing an arbitrary other value)
- Result in a segmentation fault / core dump

Unfortunately, programmers can run easily into this problem. One recent real-world example by one of the authors of this paper is the following:

Improving the following code:

```
struct Person {
    std::vector<int> values;
    ...
};

for (auto elem : createPerson().values) { // OK (lifetime extended)
```

by introducing a getter to have better encapsulation, suddenly caused the undefined behavior:

```
class Person {
private:
    std::vector<int> values;
    ...
public:
    const auto& getValues() const {
        return values;
    }
};

for (auto elem : createPerson().getValues()) { // UB (fatal runtime error)
```

The Root Cause for the problem

The reason for the undefined behavior above is that according to the current specification, the range-based for loop internally is **expanded to multiple statements**:

- First, we have some initializations using the *for-range-initializer* after the colon and
- Then, we are calling a low-level for loop

For example, the following call of the range-based for loop:

```
for (char c : createStrings().at(0)) ... // UB (fatal runtime error)
```

is defined as equivalent to the following:

```
auto&& rg = createStrings().at(0); // doesn't extend lifetime of returned vector
auto pos = rg.begin();
auto end = rg.end();
for ( ; pos != end; ++pos ) {
    char c = *pos;
    ...
}
```

And the following call of the loop:

```
for (int i : createOptInts().value()) ... // UB (fatal runtime error)
```

is defined as equivalent to the following:

```
auto&& rg = createOptInts().value(); // doesn't extend lifetime of returned optional
auto pos = rg.begin();
auto end = rg.end();
for ( ; pos != end; ++pos ) {
    int i = *pos;
    ...
}
```

And the following call of the loop:

```
for (int i : createPerson().getValues()) ... // UB (fatal runtime error)
```

is defined as equivalent to the following:

```
auto&& rg = createPerson().getValues(); // doesn't extend lifetime of returned Person
auto pos = rg.begin();
auto end = rg.end();
for ( ; pos != end; ++pos ) {
```

```
    int i = *pos;
    ...
}
```

By rule, all temporary values created during the initialization of the reference `rg` that are not **directly** bound to it are destroyed before the raw `for` loop starts.

Note that references **do** extend the lifetime of objects when they refer to sub-objects. That's why without using getters the example above works fine:

```
for (int i : createPerson().values) ... // OK (extends lifetime of returned Person object)
```

So **using member functions instead of members introduces lifetime problems.**

Severity of the problem

This **is** a serious problem for the following reasons, which we will explain in details in the section:

- The problem was raised several times by multiple people.
- Programmers run into this problem in practice.
- The problem creates significant drawbacks to teach C++.
- The range-based `for` loop becomes a loop style guides more and more warn about.
- Useful API's are not provided due to the danger of this problem.
- The problem reduces the credibility of C++.

A use of the range-based `for` loop (without using the optional `init`-statement) looks like **one statement without any semicolons to signal any end of a lifetime** (as we have to signal lifetime issues with `init`-statements).

Therefore, the problem of the range-base `for` loop is not obvious for its users:

- The ordinary programmer has the impression to be able to iterate safely over all members of the range on the right of the colon.
- Even experienced C++ programmers struggle to see the problem.

This is not theory. This is confirmed by various programmers I talked with and taught ("*oh, **that's why my code was broken***").

But, we teach this as **the loop** to use to iterate over all elements (because of its simplicity as we e.g. can't pass a wrong size). So, beginners and even advanced programmers do not see/know that there is a hidden problem in the definition of the loop so that possible code might not work.

Unless a programmer knows all the details of the definition of the loop (including rule for lifetime extensions, universal/forwarding references) it is not obvious that the loop is specified in a way that

- a) references are internally used that
- b) might limit the lifetime of some of the objects of the expression after the colon.

The loop as a whole acts as one statement and there is no signal in the use of the loop there is a hidden lifetime problem (such as having a semicolon). The average programmer is not aware of the problem. But even worse, the code might run until it gets into production.

As a consequence of the non-obvious problems of the loop,

- **We have to warn about the use of the range-based `for` loop,**

And:

- We have to explain how the range-based `for` loop is implemented and what this means
 - Show how the range-based `for` loop is defined in detail
 - Explain references
 - Explain `auto&&`
 - Explain the lifetime extension rules of references in detail

For example, Nicolai Josuttis teaches the loop (to beginners) as follows:

Range-Based for Loop Caveats

- The range-based for loop does **not work** when iterating over references to temporary objects

```
std::vector<std::string> returnValues(); // forward declaration

for (auto elem : returnValues()) { // OK
    ...
}

for (auto elem : returnValues().at(0)) { // fatal runtime error (undefined behavior)
    ...
}

// assume we know that the returned vector cannot be empty:
for (auto elem : returnValues()[0]) { // fatal runtime error (undefined behavior)
    ...
}
```

Don't use nested function calls behind the colon

There are already **style guides** that mark the use of the range-based for loop as unsafe:

- See for example the categorization of the range-based for loop as “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).

And due to the flaws revealed by this loop (and as consideration of lifetime issues in C++ in general), we can't implement:

```
for (int val : getVector() | rng::views::transform(...))
    | rng::views::filter(...)) { // doesn't compile
    ...
}
```

While the following works (even when declaring vec as reference):

```
auto vec = getVector();
for (int val : vec | rng::views::transform(...))
    | rng::views::filter(...)) { // OK
    ...
}
```

As another example for restrictions caused by this problem consider using `std::as_const()` in a range-based for loop:

```
std::vector<int> vec;
for (auto&& val : std::as_const(getVector())) {
    ...
}
```

Both `std::ranges` with operator `|` and `std::as_const()` have a deleted overload for rvalues to disable this and similar uses. With the proposed fix things like that could be possible. We can definitely discuss the usability of such examples, but it seems that there are more example than we thought where the problem causes to `=delete` function calls for rvalues.

Why is the Severity higher than in other places with dangling references?

You might argue that even with a fix we can still get easily into trouble with

```
const auto& str = get<0>(getTuple());
```

or

```
const std::string& str = getStrings()[0];
```

or

```
auto&& sp = std::span(getValues().data(), 5);
```

or

```
for (const auto& v = getColl().getValue(); auto elem : v)
```

However, the key difference to the problem inside the range-based `for` loop is that in all these examples **the risk is visible**, because usually two visible things are necessary to have a potential risk with references to deleted temporary objects:

- A reference declaration (`&` or `&&` or `decltype(auto)`)
- A semicolon (`;`)

The range-base `for` loop is **the only place** in the C++ standard where this problem occurs without a visible reference and without a visible semicolon. **The fact that one statement internally breaks into multiple statements so that only *some* of the involved objects have a lifetime until the end of the statement.**

And the discussion of <http://wg21.link/cwg900> agrees:

This [fix] also removes **the only place** where binding a reference to a temporary extends its lifetime implicitly, **unseen** by the user.

Proposed Solution

We propose to fix this problem of the range-based `for` loop by a modification of the way the loop is specified. The internal initialization of the range as universal/forwarding reference shall no longer act as a separate statement that ends lifetimes before the internal `for` loop is entered.

Taking the last motivating example, a statement such as

```
for (auto elem : foo().bar().getValues()) ...
should be expanded equivalent to the following:
auto&& tmp1 = foo();           // lifetime of all temporaries extended
auto&& tmp2 = tmp1.bar();      // lifetime of all temporaries extended
auto&& rg = tmp2.getValues();
auto pos = rg.begin();
auto end = rg.end();
for ( ; pos != end; ++pos ) {
    auto elem = *pos;
    ...
}
```

The question is how to formulate that without introducing new (lifetime) rules for C++.

We could:

- a) Use black-box wording for this special case
- b) Use the current wording and state that the internal initializations in the definition of the range-based `for` loop are not statements that end the lifetime of subexpressions in the `for`-range-initializer
- c) Come with a different `as-if` clause using a lambda

For example, one idea for the wording of the fix was to use a lambda:

```
[&](auto&& rg) {
    auto pos = rg.begin();
    auto end = rg.end();
    for ( ; pos != end; ++pos ) {
        auto elem = *pos;
        ... // special return, goto, co_yield, co_return handling
    }
}(foo().bar().getValues()); // all temporary return values valid until the end of the loop
```

See <https://wandbox.org/permlink/KRecfQhE696LD4DC> for an example without and with this fix.

Because the expression we initialize `rg` with is now no longer a separate statement, the lifetime of all prvalues returned in the expression that is the initial range remain valid until the end of the whole loop.

However, in that case we have to specify special handling for `return`, `goto`, and `co-routine` statements because they have to leave the scope where the loop is called.

For simplicity, we propose the wording of option a). This way to specify the special behavior of the loop also has the advantage that we do not have to reveal a new general language feature or lifetime rule (beside specifying the new feature here).

The concrete proposal is to propose a fourth special rule about the lifetime of temporaries. and propose the following change. change with **option a)** might look as follows:

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.

Q&A

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

This *is* a problem we see in practice.

For example:

- Official standards (such as for safety critical systems) warn about or even restrict the use of the range-based `for` loop.
- At a major IT company the internal development platform has multiple posts of lifetime problems with the range-based `for` loop.
- One authors of this paper personally run into this problem just recently when replacing a direct member access to using a getter (as described in the section [Unfortunately.](#)).
- Designs are already changed to avoid the possibility to run into this problem.
- Style guides warn already about the range-based `for` loop.
- The internet has discussions about this issue. For example:
 - <https://stackoverflow.com/questions/51436155/range-based-for-loop-on-a-temporary-range>
 - <https://softwareengineering.stackexchange.com/questions/262215/who-is-to-blame-for-this-range-based-for-over-a-reference-to-temporary/262243>

It is for sure a **significant problem in teaching**, because you either have to constrain the use without explanation or show details of the definition of the range-based `for` loop understandable only by experts. It's surprising how many developers are surprised by this issue even in advanced trainings.

Finally, with new classes and functions with reference semantics, we get more and more problems like this.

Consider the motivating example with `std::optional` (C++17) and using `std::span` (C++20) as follows:

```
for (auto elem : std::span{getColl().data(), 5}) { // UB (fatal runtime error)
    ...
}
```

This lowers the credibility of C++, especially for beginners.

Research for possible broken code also revealed that this problem makes code already unnecessary complex.

Is there code that might be broken with the fix?

The proposed fix would extend the lifetime of some objects. This can be an issue if the extension of lifetime of an object in the header of the loop conflicts with action done inside the loop.

For example:

```
for (auto elem : lockedAccess{obj, objMx}->getData()) {
    ...
    lockedAccess{obj, objMx}->getName() // deadlock with the proposed fix
}
```

However, look how fragile this code is. It only works if `getData()` really yields a value. The same code leads to a fatal runtime error if `getData()` yields a reference because we now can use the referred value and the name concurrently:

```
for (auto elem : lockedAccess{obj, objMx}->getRef()) {
    ...
    lockedAccess{obj, objMx}->getName() // UB with traditional range-based for loop
}
```


So, how much code is broken in practice?

We don't expect that we will find broken code as above in practice.

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

*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.

The person doing the research 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.*

Can compilers warn about possible 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

We do not expect many false positives.

Is there a performance penalty?

For sure, extending the lifetime of an object might cause that we temporarily need more memory while the loop is running.

A benchmark testing the impact on speed demonstrates not significant difference:

See <http://quick-bench.com/q/iIOq7qDEoE-ZmB29XzWa2fDBUPk>

But don't we have the same problem with initializers in loops?

You can argue that we have the same problem in code like this:

```
for (const auto& v = createPerson().getValues(); !v.is_empty(); ...) ...
```

However: In this code there is a **significant difference** to the problem raised: Programmers can see that there might be a critical partial lifetime extension:

- **a semicolon signals the end of a statement**
- **a reference is used**

That is, at all other places, we usually have two signals for possible problems.

In code like

```
for (int i : foo().bar()) ... // OOPS: different lifetime extensions in one statement
```

the programmer has to know that **the lifetime of objects extends differently inside the same expression** in a context where there is no signal for an end of a statement. We are not aware of any other location in the C++ standard where we have a situation like this. And the discussion of <http://wg21.link/cwg900> agrees:

This [fix] also removes **the only place** where binding a reference to a temporary extends its lifetime implicitly, **unseen** by the user.

Programmers tend to read the code above as being no more dangerous than performing a nested function call:

```
loopOver( foo().bar() ); // OK, looks like equally safe
```

But don't we break the zero-overhead principle?

Programmers should not pay for things they don't need. Usually this means, code from using a feature should not be slower or bigger than code not using this feature. However, when the risk of a problem is severe and we have a simple way **not** to get the overhead, we prefer safety (especially for features for non-experts). For example:

- We pass parameters to threads by value (unless otherwise specified)

And if it is worth it, we even change C++ accordingly. For example:

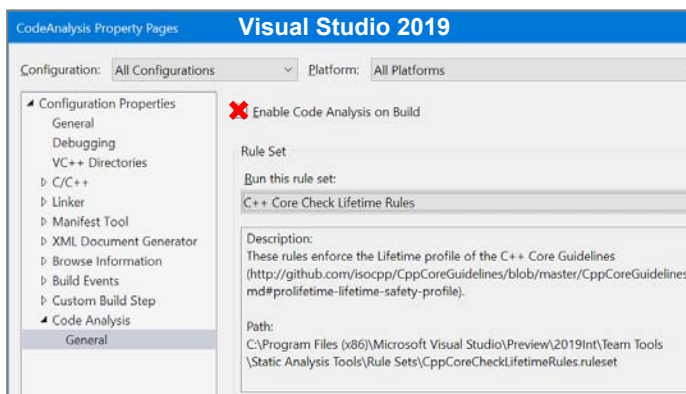
- We introduced a defined evaluation order for additional operators in C++17

This fix clearly falls into this category: We avoid severe errors (undefined behavior) and we have easy workarounds if it might introduce a performance issue.

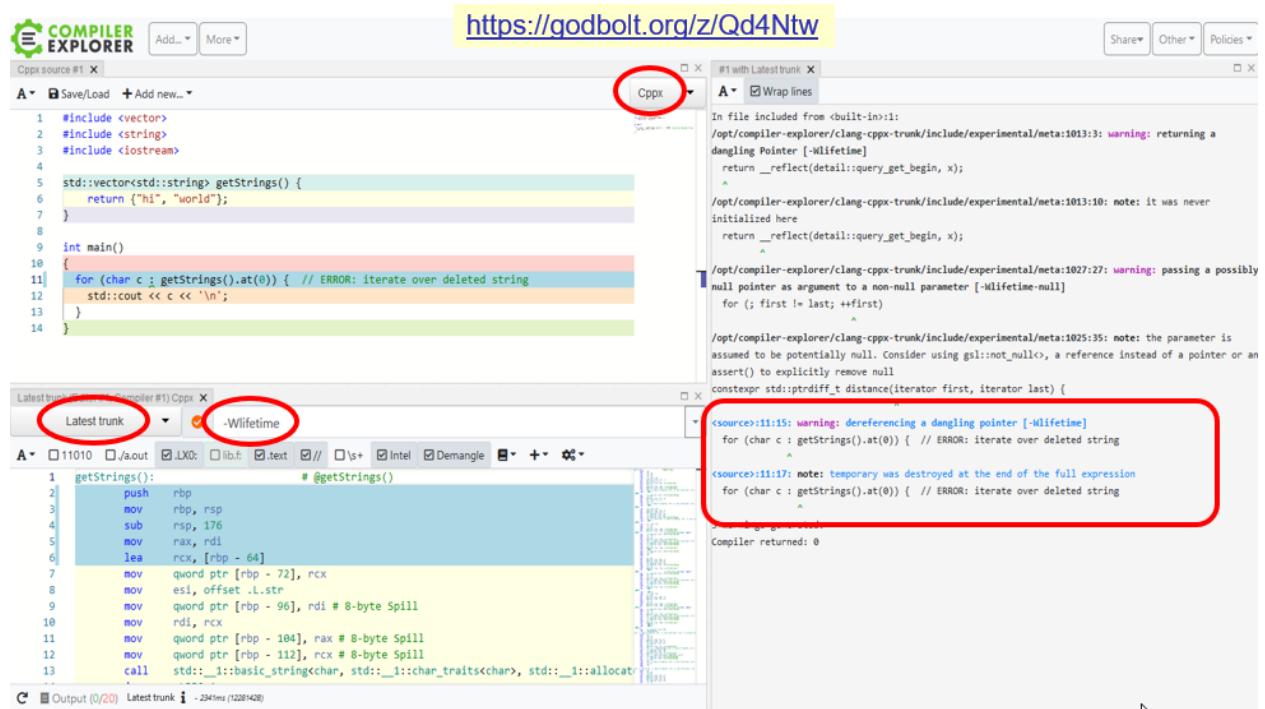
Note especially that the range-based `for` loop is **not** a low level feature. It is already a layer on top of the basic `for` loop. **Such a layer should have fewer security risks, not more.**

But don't we have tools to detect such lifetime problems?

The examples in this paper are **partially** diagnosed by Herb Sutter's [Lifetime rules spec](#) in the C++ Core Guidelines, which is now partially part of clang and can be used in Visual Studio as follows:



In fact, the first two examples get useful diagnoses with this extension. You get messages like the following:



See

- <https://godbolt.org/z/Qd4Ntw>
- <https://godbolt.org/z/76v4v6>

However,

this lifetime extension does not solve the general problem raised here for the following reasons:

- These lifetime extensions are not standardized yet and need special compiler support.
- When not using standard library types (such as the motivating type `Person`), you have to specify the corresponding lifetime dependencies. This means that for each and every getter (and other function) returning a reference to a member, we have to provide the corresponding lifetime dependencies for each and every static-analysis tool.
- We still have to explain why we get the error and how to avoid it.
- Unfortunately, the lifetime extension only gives a warning, not an error and many projects ignore warnings (due to the problem that several warnings are not severe and that we have too much).

And making it an error would create false positives. For example:

```
class Person {
    std::string name
public:
    const std::string& getName() const {
        return name;
    }
    const std::string& getAnswer() const {
        static std::string s{"42"};
        return s;
    }
};

for (char c : getPersonByValue().getName()) // ERROR
for (char c : getPersonByValue().getAnswer()) // OK
```

If the getters are not inline defined, a compiler has no way to find out that one usage of it in the range-base for loop is safe, while the other is not.

The programmer would have to instrument the compiler somehow (or we introduce a syntax to specify lifetime dependencies or lifetime dependency exceptions).

Thus, the lifetime extensions helps to lower the severity of the consequences of this problem. But still we have the problem and have to warn about using the range-based `for` loop, explain why, and discuss alternatives.

But don't we have a solution (such as a new loop) without the risk of breaking existing code?

The current behavior is fundamental trap in the most used control structures of C++. Programmers not being aware of the problem will continue to run into this trap (and not use and any workaround if provided).

A workaround only helps if we disable or deprecate the existing range-base `for` loop so that a programmer has to react. We don't think that deprecating the range-based `for` loop is appropriate.

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

Since C++20, we can avoid the problems of the range-based `for` loop by using the range-based `for` loop with initialization (see <http://wg21.link/p0614>):

Instead of

```
for (char c : createStrings().at(0)) ... // UB (fatal runtime error)
```

we can implement

```
for (auto rg{createStrings()}; char c : rg.at(0)) ... // OK
```

or

```
for (auto&& rg{createStrings()}; char c : rg.at(0)) ... // OK
```

However,

this new language feature does not solve the problem raised here for the following reasons:

- We still have to explain why we get the error when using the ordinary range-based `for` loop.
- The code becomes significantly more clumsy.
- We have to teach lifetime extension of references to understand this workaround.

In fact, without careful comments, programmers might even "improve" the workaround by turning it into code using the ordinary range-based `for` loop (not knowing that they introduced a severe runtime error).

Shouldn't we fix lifetime extension for references in general?

Instead of fixing the range-based `for` loop, we could also change the general rules for extending the lifetime objects references refer to.

This was proposed earlier a couple of times for C++, but it was always rejected for good reasons. To quote Bjarne Stroustrup here:

Way back in the 1980s, the lifetime of an object declared in the `for`-initializer extended to the end of the block (you can find that in the ARM). Initially, I thought that safer, but changed my mind after many complaints. The most serious was the long life of matrix temporaries that basically rendered initialization in `for`-statements useless. Be very careful when trying to extend lifetimes; the results can be surprising and costly.

When the issue was discussed as EWG issue 120 there were also significant concerns about introducing a new lifetime model as well as extending lifetimes in general due to extended memory footprint (see <http://wiki.edg.com/bin/view/Wg21rapperswil2014/EvolutionWorkingGroup>).

It might be great to solve lifetime issues in some more general way, but that doesn't mean we couldn't/shouldn't solve this one directly.

Note that this problems only occurs due to **the way** we specify the behavior of the range-based `for` loop. If we would provide the behavior more like a black box of function call, we wouldn't have this problem and the loop would match the expectations of the programmers.

Are there other places in the language that have similar (theoretical or real) problems?

Not that we are aware of.

Again, as the discussion of <http://wg21.link/cwg900> states:

This [fix] also removes **the only place** where binding a reference to a temporary extends its lifetime implicitly, **unseen** by the user.

See [Why is the Severity higher than in other places with dangling references?](#) for details.

What are the drawbacks of such a fix?

A fix like this would usually not break existing code, because we would extend just the lifetime of a few temporary objects a bit longer (yes, there are ways that such an extensions breaks functional behavior with very subtle programming).

Only if the lifetime extension is a problem (such as when a subexpression of the initialization temporarily holds a lock and that lock is extended), we might get into trouble. For example:

```
for (auto elem : lockedAccess{obj, objMx}->getValues()) {
    ...
}
```

With a type such as `boost::synchronized_value` even deadlocks might occur:

```
boost::synchronized_value syncObj{obj};
for (auto elem : syncObj->getValues()) {
    if (...elem...)
        syncObj->getName() // deadlock with the fix
}
```

However, note that this code above turns into a data race, when using references instead:

```
boost::synchronized_value syncObj{obj};
for (const auto& elem : syncObj->getValues()) {
    if (...elem...) // data race even without the proposed fix: unsynchronized access to obj
        syncObj->getName() // deadlock with the fix
}
```

The question is, how likely code is where we iterate over copies of multiple elements of an object and then inside deal with the object again and knowing that the whole code is broken if the declaration of an element uses a reference instead (which might easily happen accidentally if others see this code).

We consider this close to a pathologic example,

We also see no ABI break because object code compiled with the old behavior could coexist with object code having the new behavior.

Regarding performance. First, running time should not be affected. We would only extend the lifetime of `pvalues` in initializers of the range-based `for` loop until the end of the loop. That is, we only delay a destruction to a later timepoint but do not execute additional code.

However, when temporary objects live longer than expected, programs might temporarily need more memory. That is, when a function returns an expensive value, the resource is hold while possible additional resources are used. This might in rare cases extend memory limits.

Note that **usually** (in all cases where we just have one expression on the right side of a range-based `for` loop, **the proposed change would have no effect at all**).

Note also that a programmer still can avoid any overhead of the range-based `for` loop by using the optional initializer of the loop or using an ordinary `for` loop.

What was discussed about this problem before?

This problem was raised and discussed a couple of time. Unfortunately, we never got a resolution.

Core issue 900 (<http://wg21.link/cwg900>) and core issue 1498 (<http://wg21.link/cwg1498>) raised exactly this problem in 2009 and 2012, which then 2014 became EWG issue 120 (<http://wg21.link/ewg120>).

The comment by CWG on issue 900 was:

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.

The comment in EWG was:

Discussed in Rapperswil 2014. **EWG wants a solution**, and welcomes a paper tackling the issue. Vandevorde 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?

We want to point out that **the proposed solution covers all concerns raised in EWG**:

- We don't introduce a new lifetime model for C++. We only change the guarantees the range-based `for` loop gives to programmers.
- We don't modify the end-of-full-expression rule.

Proposed Wording

(All against N4878)

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] ad 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:

Affected subclause: 6.7.7 [also 8.6.5] [class.temporary] and [stmt.ranged]

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

Rationale: Because when the range-base-initializer is a reference to a temporary object, the loop operates on destroyed objects.

Effect on original feature: The lifetime of a temporary object in the *for-range-initializer* might be extended until the end of the range-based `for` loop.

[Example1:

```
    for (auto e : getValue().getRef()) { // lifetime of getValue() extended
    ...
    } // until here
-- end example]
```

Feature Test Macro

Provide a new value for `__cpp_range_based_for`

Acknowledgements

Thanks to a lot of people who discussed the issue, proposed information and possible wording. Especially: Jens Maurer, Herb Sutter, Tim Song, Antony Polukhin, Barry Revzin, Ville Voutilainen, Bjarne Stroustrup, Titus Winters.

Forgive me if I forgot anybody.