Project: ISO JTC1/SC22/WG21: Programming Language C++
Doc No: WG21 **P0936R0**
Date: 2018-02-12
Reply to: Richard Smith (richardsmith@google.com), Nicolai Josuttis (nico@josuttis.de)
Audience: EWG, CWG
Prev. Version:

Bind Returned/Initialized Objects to the Lifetime of Parameters, Rev0

Lifetime issues with references to temporaries can lead to fatal and subtle runtime errors. This applies to both:

- Returned references (for example, when using strings or maps) and
- Returned objects that do not have value semantics (for example using std::string_view).

This paper proposes a new language feature to detect and avoid these errors.

Motivation

Let's motivate the feature for both classes not having value semantics and references.

Classes not Having Value Semantics

C++ allows the definition of classes that do not have value semantics. One pretty new but already famous example is **`std::string_view`**: The lifetime of a string_view object is bound to an underlying string or character sequence.

Because string has an implicit conversion to string_view, it is easy to accidentally program a string_view to a character sequence that doesn't exist anymore.

A trivial example is this:

```
std::string_view s = "foo"s;   // fatal runtime error
```

This can occur more indirectly as follows:

```
std::string operator+ (std::string_view s1, std::string_view s2) {
    return std::string{s1} + std::string{s2};
}

std::string_view sv = "hi";
sv = sv + sv;    // fatal runtime error: sv refers to deleted temporary string
```

If you argue that the bugs in these examples are easy to see, consider a template calling operator+ instead for a passed string view:

```
template<typename T>
T add(T x1, T x2) {
    return x1 + x2 ;
}
sv = add(sv, sv);   // fatal runtime error
```

We already see code like this.

We can better support these types by giving the ability to annotate a function declaration to indicate that the lifetime of an object used as part of a parameter's value or the `*this` object must live at least as long as the return value does.

For example, the problem above can be detected or even just made to work if the constructor defining the implicit conversion is marked so that the lifetime of the created string_view depends on a passed string.

- If the implicit conversion is enabled by a constructor, this might look as follows (with ***lifetimebound*** representing whatever we might introduce as new feature):

```
template<class charT, class traits = char_traits<charT>>
class basic_string_view {
 public:
  …
  template<class Allocator>
  basic_string_view(const basic_string<charT, traits, Allocator>&
                            str lifetimebound) noexcept;
  …
};
```

- If (as it is the case currently) the conversion is defined by a conversion operator, this might look as follows:

```
template<class charT, class traits = char_traits<charT>,
                      class Allocator = allocator<charT>>
class basic_string {
 public:
  …
  operator basic_string_view<charT, traits>() const lifetimebound noexcept;
  …
};
```

Note that we could use the feature to

- Either extend the lifetime of prvalues as it is done for references to prvalues.
- Or enable compilers to warn about code like in the example above.

Returned References to Temporaries

Similar problems already exists with references.

A trivial example would be the following:

```
struct X { int a, b; };
int& f(X& x) { return x.a; }      // return value lifetime bound to parameter
```

Class std::string provides such an interface in the current C++ runtime library. For example:

```
char& c = std::string{"hello my pretty long string"}[0];
c = 'x';                          // fatal runtime error
std::cout << "c: " << c << '\n'; // fatal runtime error
```

Again, we should be able mark the return value of the index operators/functions accordingly to get corresponding warnings or the expected behavior:

```
template<class charT, class traits = char_traits<charT>,
                      class Allocator = allocator<charT>>
class basic_string {
 public:
  …
  const_reference operator[](size_type pos) const lifetimebound;
  reference operator[](size_type pos) lifetimebound;
  const_reference at(size_type n) const lifetimebound;
  reference at(size_type n) lifetimebound;
  …
};
```

There are more tricky cases like this. For example, when using the range-base for loop:

```
for (auto x : reversed(make_vector()))
```

with one of the following definitions, either:

```
template<Range R>
reversed_range reversed(R&& r) {
```

```
  return reversed_range{r};   // return value lifetime bound to parameter
}
```

or:

```
template<Range R>
reversed_range reversed(R r) {
  return reversed_range{r};   // return value lifetime bound to parameter

}
```

Again, we could either make this code work or warn about dangling “references” by marking the return value as lifetime dependent from the parameter.

Finally, such a feature would also help to detect or fix several bugs we see in practice:

Consider we have a function returning the value of a map element or a default value if no such element exists without copying it:

```
const V& findOrDefault(const std::map<K,V>& m, const K& key,
                                               const V& defvalue);
```

then this results in a classical bug:

```
std::map<std::string, std::string> myMap;
const std::string& s = findOrDefault(myMap, key, "none"); // runtime bug if key not found
```

Again, the marker would help one way or the other:

```
const V& findOrDefault(const std::map<K,V>& m, const K& key,
                                               const V& defvalue lifetimebound);
```

Lifetime Extension or Just a Warning?

We could use the marker in two ways:

1. Warn only about some possible buggy behavior.
2. Fix possible buggy behavior by extending the lifetime of temporaries.

A lifetime extension will also fix an asymmetry we currently have between built-in language constructs and that what can be achieved by user-defined types and functions. For example with

```
struct A {
  std::initializer_list<int>&& x;
  const int& y;
};
```

We have lifetime extension for an A temporary, an initializer_list temporary, a const int[3] temporary holding the value of the initializer list object, and an int temporary:

```
A&& a = A{ {1, 2, 3}, 50 };
```

But it is not possible to write a constructor for A that has the same effect. Likewise:

```
struct X { int n; };
struct Y : X {};
struct B { Y y; };
int&& r = static_cast<X&&>(B{3}.y).n; // reference lifetime-extends complete B object
int&& s = implicit_cast<X&&>(B{3}.y).n; // impossible to define implicit_cast so that this
                                        // lifetime-extends properly
```

However, we have to be careful. If we make some cases automatically extend lifetime, but leave other cases not doing so, lifetime bugs will become less obvious (you cannot see lifetime bugs in the caller and need to consult the callee) and so we risk them becoming more common due to people not thinking about temporary lifetime issues any more.

We have no decision what to propose here and need some further discussion about it.

Where to Apply the Feature?

Another decision to make is whether this feature should apply to primitive types, pointers, etc. Consider for example:

```
using Array = int[3];
int* p = Array{1, 2, 3};          // lifetime-extend array through array-to-pointer conversion?
int& r = *(int*)(intptr_t)&(int&)(int&&)42;  // lifetime-extend through pointer-to-integer cast?
```

Without lifetime extending here, we can get the following inconsistency:

```
intptr_t get_addr(int&& r lifetimebound) {
  return (intptr_t)&r;
}
intptr_t n = get_addr(42);        // #1, does lifetime-extend
intptr_t m = get_addr(42) + 1;    // #2, does not lifetime-extend?
```

For consistency, perhaps we should say that #1 does not lifetime-extend either and that lifetime bounds can only be applied on function parameters for functions that return a class or reference types or (for constructors) initialize an object.

Alternatively, we could allow lifetime to be transported through all of the above, by only "blocking" the lifetime inference at lvalue-to-rvalue conversions, discarded-value expressions, and function parameters without the ***lifetimebound*** marker.

Name and Form of the Marker

Last not least, we have to decide about the way we mark:

- Use an existing keyword such as "return" or "export"
- Use a new keyword
- Use a context-sensitive keyword, and require the parameter to have a name
- Use an attribute
    - But if we want to extend the lifetime and follow our normal rules for attributes, then this can't be an attribute, because it changes program semantics in observable ways.

We are slightly leaning towards a new, context-sensitive keyword.
Of course, we also need a name for it, then. Possible options might be something around the area of “reference semantics”, “lifetime extension”, or “depending”. The word should not be too long but also no common parameter name. For example

- referenced
- bound
- lifetimebound
- …

Proposed Wording

Here, for the moment, we propose the wording to allow lifetime extension to be propagated through function calls. No wording is provided for how to specify the marker yet, as the precise syntax for the marker is an open question.

Change only **[class.temporary] paragraph 6** as follows:

Replace the first sentence as follows:

> The third context is when a ~~reference is bound to a temporary object~~.[116]

By:

> The third context is when a temporary object is *lifetime-bound* to the initializer of a declared variable, as follows:
>
> - A temporary object is lifetime-bound to the expression corresponding to its temporary materialization conversion (7.4).
> - All temporary objects that are lifetime-bound to an expression E are also bound to an expression X of which E is an immediate subexpression if no conversions are applied to E other than the temporary materialization conversion, and either X results in the invocation of a function (including a constructor) where the parameter or implicit

object parameter corresponding to E is marked as ***lifetimebound***, or X does not result in the invocation of a function and has one of the forms
— (E),
— E [E2] or E2 [E], where E is of array type,
— E . E2, where E2 designates a non-static data member of non-reference type,
— E .* E2, where E2 is a pointer to data member of non-reference type,
— const_cast<T>(E), static_cast<T>(E), dynamic_cast<T>(E), or reinterpret_cast<T>(E),
— *simple-type-specifier*(E) or *typename-specifier*(E),
— *simple-type-specifier braced-init-list* or *typename-specifier braced-init-list*, where E is a constituent expression of the *braced-init-list*,
— (*type-id*) E,
— E1 ? E : E2,
— E1 ? E2 : E, or
— E1 , E.

- A temporary object is lifetime-bound to the expression corresponding to its temporary materialization A temporary object is lifetime-bound to an initializer if it is lifetime-bound to any constituent expression of that initializer.

What happens with the footnote?

And replace the second sentence as follows:

The temporary object ~~to which the reference is bound or the temporary object that is the complete object of a subobject to which the reference is bound~~ persist~~s~~ for the lifetime of the reference if the glvalue to which the ~~reference is bound was obtained through one of the following:~~
~~—~~ ...
~~—~~ ...

By:

The temporary objects that are lifetime-bound to the initializer of the variable persist for the lifetime of the variable.

And fix the examples and notes 6.9/6.10/6.11 there as follows:

~~The exceptions to this lifetime rule are:~~
[Note:
A temporary object bound to a reference parameter in a function call (8.5.1.2) persists until the completion of the full-expression containing the call if the parameter is not marked as ***lifetimebound***.]
[Note: The lifetime of a temporary bound to the returned value in a function return statement (9.6.3) is not extended; the temporary is destroyed at the end of the full-expression in the return statement.]
[Note: A temporary bound to a reference in a new-initializer (8.5.2.4) persists until the completion of the full-expression ...]

Acknowledgements

Thanks to all who contributed to the discussion of this feature.

Feature Test Macro

This language feature should have the following language feature macro (even if it only results in a warning, because source code looks different with this feature. The proposed macro is:

```
__cpp_lifetime_bound
```