Document number: P0066
Date: 2015–09–28
To: SC22/WG21 EWG
Revises: N4221
References: N3918, N0345
Reply to: David Krauss
(david_work at me dot com)

Accessors and views with lifetime extension

Abstract

A new model associates an expression with the set of temporary objects needed to represent its value. This enables lifetime extension to work transparently through accessor functions which opt in using the `export` keyword. The same mechanism enables view classes and adaptors to retain an underlying container. The current lifetime extension behavior is also more elegantly described, and the "temporary expression" controversy is resolved. This solves EWG 120 and many other dangling reference issues. When lifetime extension is impossible, lifetime analysis is improved to enable diagnosis. No dedicated lifetime analyzer is needed; `export` annotations take only three distinct values. Full polymorphic lifetime analysis is left open to future development. Since the `export` keyword marks accessor functions, it also provides a notation to eliminate the boilerplate associated with `const`- and rvalue- (`&&`) qualified overloads.

1. Background

Lifetime extension applies when a reference is bound to a temporary object or its subobject. Unfortunately, this rule has corner cases ranging from difficult to undecidable. Fundamentally it requires tracking lvalues through expression evaluation, performing a runtime task at compile time. In historical practice, lifetime extension only worked when initializing a reference by a prvalue expression, or a class member access with a prvalue object operand. Since C++11, `std::initializer_list` additionally behaves as if it were a reference-to-array type.

More recently, N3918 (Maurer, 2014) extends the range of suitable expressions to include explicit casts and array subscripts. (It has not been formally adopted, but it is widely implemented. Such extensions are within the latitude of the standard.)

This proposal takes the direction of N3918 to its limit. Not only casts, but any expression that can obtain a reference to a prvalue may provide lifetime extension. Not only array subscripts, but subscript operator overloads work.

Additionally, reference-like behavior as implemented by `initializer_list` is opened to the user. The focus shifts to view and range classes, not the container that manages underlying resources. Libraries are enabled to create wrappers that appear to own non-member objects on the stack. `initializer_list` itself ceases to be a special case of core support, outside of type deduction. This potentially opens a similar approach to arrays of runtime bound.

Compared to the previous revision, N4221:

- Strongly typed semantics are added for the qualifiers that denote accessor functions. This parallels the ongoing removal of the underlying classic exception-specification semantics in N4533 (Maurer, 2015).
- The rules for determining value ownership are simplified.
- Finer annotations differentiate classes that actually own values from intermediaries such as iterators, which merely help to access values.
- Syntactic sugar is added for defining member accessor functions that propagate `const` qualification and rvalue category from the object expression.
- The supporting material has been heavily revised.

1.1. Origins of lifetime extension

Lifetime extension is a very old C++ feature, originating before ISO standardization began. Initializing a `const&` reference variable from a function result to avoid a copy has worked since references were introduced. It has two ingredients: binding an rvalue to a `const&`, which is needed for operator overloading, and assuming that that the temporary lives to the end of the block. Temporaries all worked that way at first, a rule which was only reevaluated when scalability reared its ugly head. (The term “temporary” originally had no temporal connotation.) During the drafting of the ARM (1988-1989), Stroustrup blessed the idea of early destruction of

temporary objects [1]. Compilers then used different rules for temporary lifetimes, trading off safety against efficiency. Basic interoperability required some form of lifetime extension rule.

The first paper to consider lifetime extension was N0345 (Bruns, 1994). It recommended consistency between pass-by-reference argument subexpressions and named references used as arguments. The proposed rule was broad:

> For any statement explicitly binding a reference to a temporary, the lifetime of all temporaries in the statement are extended to match the lifetime of the reference.

Presumably this was considered to extend too many lifetimes, leading to subtle problems. The next working paper introduced the current system.

1.2. Status quo

Lifetime extension is already more flexible than often assumed. When a prvalue expression, or an lvalue expression accessing a subobject of a prvalue, initializes a reference, the entire temporary object backing the prvalue receives the storage duration of the reference. So, the reference may bind to a slice of the actual object. If the temporary has member references, the rule applies recursively to them.

```
struct b { int m; };
struct d : b { ~ d() { std::cout << "~\n"; } };
struct c { d const & r; };

int main() {
    auto && whole = d{};                    // Extend by binding object to a reference,
    b && base = d{};                           // or base subobject to a reference,
    int && member = d{}.m;                  // or member subobject to a reference,
    c aggregate = { d{} };                     // or object to member reference,
    std::initializer_list< c > il = { d{} };// or to reference inside an IL element.
    std::cout << "wait for it\n";
} // Five d::~d() calls occur here.
```

This implements a kind of static type erasure: An invisible, perhaps inaccessible object provides underlying functionality to an interface handle. The most familiar application of this technique is the C++03 scope guard idiom [2]. C++11 `auto` type deduction has obviated the need for a scope guard base class, but other applications exist. With some compiler help, `initializer_list` follows the same pattern: The invisible object is the underlying array, the functionality is the provision of storage, and the interface handle is `initializer_list`. Without lifetime extension by slicing, the array size would need to be part of a user-visible type, i.e. as a template argument. This design pattern has the potential to reduce template bloat by allowing view-adaptor classes to avoid encapsulating the underlying object. (See below for deeper analysis.)

---

[1] Design and Evolution of C++, Stroustrup, 1994, §6.3.2 (p143). ARM stands for *The C++ Annotated Reference Manual*, the initial informal standard, not Acorn RISC Machine.

[2] *Change the Way You Write Exception-Safe Code — Forever*, Alexandrescu and Marginean, 2000.

1.3. Temporary expressions

The current specification of lifetime extension is subject to several defect reports, clustered around CWG DR 1299, with a tentative resolution in N3918. The current general strategy in [class.temporary] §12.2/5 [3] requires statically resolving the referent object initializing a reference to see if it is temporary, but excludes objects which don't appear to be propagating to the full-expression value, such as function arguments. Expressions which may propagate lifetime extension are temporary expressions. The problem is that such propagation may be specified without using anything so suspicious.

```
struct self_ref { self_ref & me = * this; };        // no reference parameter…
self_ref & q = self_ref{}.me;            // … but still no lifetime extension in practice
```

In this example, the object is bound to the implicit `this` parameter of its own constructor, which could be construed as a disqualification, but it this is not the intended specification nor the practice. A similar example without a nonstatic data member initializer shows implementation variance:

```
struct ref { foo const & f; };                          // No constructor at all.
foo const & q = ref{ foo{} }.f;          // Lifetime extension for ref or only foo?
```

Given this example, EDG applies lifetime extension to the value of `foo{}` but not to the `ref{…}`. Clang and GCC apply no lifetime extension.

In both examples, the initializer is a temporary expression per N3918: a class member access with a prvalue object expression. One possible resolution is that reference member accesses should not be temporary expressions. However, since binding a member reference and accessing a member each apply lifetime extension, disqualifying their combination would be inconsistent from the user's perspective.

The biggest shortcoming of the temporary expression approach is the lack of a provision for encapsulation. N3918 blesses `static_cast` and `const_cast` as temporary expressions, but a library `as_const` function can only return a dangling reference. Array subscripts are blessed, `std::array` subscripts are not.

2. Motivating examples

Most cases of accidental dangling references could be fixed by adding lifetime extension. Here are several different scenarios.

2.1. Range-based `for` statement

Dangling references are most dangerous when they are least expected. The range-based `for` statement (§6.5.4) uses an internal, anonymous reference variable bound to the *range-init* expression. This binding must perform sufficient lifetime extension to maintain the validity of the range.

---

[3] References are to the working draft N4527 unless otherwise noted.

The user may wish to apply some sort of transformation to the range. An adaptor object is used for the *range-init*, with its own iterators passing through to an underlying object. Unfortunately, attempting to compose such adaptors in a *range-init* will fail with a dangling reference, because they are all temporaries and only one can get a lifetime extension. As illustrated by EWG 120:

```
std::vector<int> vec;
for (int val : vec | boost::adaptors::reversed
                   | boost::adaptors::uniqued)
    ; // Error: result of (vec | boost::adaptors::reversed) died.
```

It is already possible to dance around this problem by some combination of aggregate initialization, and moving the containers into the adaptor.

```
template< typename underlying >
struct reversed {
    underlying u;
    using iterator = std::reverse_iterator
                         <typename std::decay_t<underlying>::iterator>;
    iterator begin(), end();
};
for (int val : reversed<reversed<std::vector<int>&&>&&>{{{1, 2, 3}}})
    ; // OK: all three temporaries receive lifetime extension.
for (int val : reversed<reversed<std::vector<int>>>{{{1, 2, 3}}})
    ; // OK: each adaptor encapsulates the underlying object.
```

The first solution requires that the adaptor has no constructor or factory function, which is onerous. The second solution requires additional move operations, which loses efficiency. In practice, encapsulation needs to happen selectively depending on the value category of the argument, which adds a dimension of template metaprogramming.

2.2. Value-selection functions

Value-semantic functions may inadvertently return prvalues by reference.

```
void foo( obj & in ) {
    int const & smaller = std::min( in.value, 5 );      // Dangling reference.
```

There is nothing conceptually wrong with either `foo` or `std::min`; the C++ language is at fault. Rather than require the user to declare a constant variable to hold `5`, the temporary object's lifetime should extend as if it was bound directly.

2.3. Refactoring argument expressions

Any good programming language should allow a long line of code to be easily broken into smaller lines. Binding a reference and passing it should be the same as passing the referent. Indeed, it's only in such a context that it makes sense to apply the term "temporary" to a referent. Instead, C++ requires the user to reason about which subexpressions generate objects responsible for significant state, and to preferentially name those objects.

2.4. Views and ranges as handles

Objects that observe the state of an external container are an important direction in the evolution of the standard library. Examples include `std::experimental::string_view` and Ranges, both of which essentially encapsulate an iterator and an end condition (which may be another iterator). Ordinary iterators and pointers fall into the same category.

Declaring a range, view, or pointer object, and initializing it from a prvalue container, guarantees a dangling reference.

```
std::string_view v = std::string( "hello" );
```

This is problematic because the only way to return content from a function is by returning a container, but the better way to inspect content is a container-agnostic handle. The concern of data storage is foisted upon the programmer. If the library wishes not to commit to a particular container type, then users must use `auto` to agnostically retain function results.

The problem is likely to get worse, but it's not new. We've all seen this beginner mistake:

```
char const * s_ptr = std::string( "hello" ).c_str();
```

Lifetime extension would prevent such headaches. The only challenge is to avoid extending temporaries that do not need it.

2.5. Return by output parameter and method chaining

In Java, it is valid to initialize an object by calling a sequence of chained methods. In C++ this only works if the object is placed on the heap.

```
Point r = new Point().setX( 5 ).setY( 42 );                    // OK in Java.
Point & r = (*new Point).setX( 5 ).setY( 42 );                 // OK in C++.
Point & r = Point().setX( 5 ).setY( 42 );    // Runtime failure, dangling reference.
```

Since C++11, the standard library fully supports rvalue iostream objects in expressions, but the resulting expression cannot be retained.

```
std::istream & f = std::ifstream( path ) >> header;     // Dangling reference.
```

Users familiar with environments that manage memory by garbage collection and reference counting tend to be surprised that C++ RAII has so little insight even within the local scope. Java, Ruby, Python, C#, and Objective-C fall into this category. Moreover, object lifetime management is a selling point and a focus in the ongoing evolution for most of these languages.

2.5.1. Expression templates with rvalue semantics

Expression template libraries benefit from conserving memory and minimizing intermediate temporaries. If temporaries could be retained as persistent storage for results, expensive copy operations could be eliminated.

A library designed this way would also be able to recycle memory from named objects passed into an expression through `std::move`.

2.6. Accessor boilerplate

Member functions that return a reference into their own object tend to be implemented with a lot of repetition, for the sake of propagating `const`-qualification. Xvalues and ref-qualifiers add a new dimension, although such overloads are seldom provided in current practice.

A function that references a value owned by the current object is always covariant with the value category of the implicit object parameter. There should be a boilerplate-free way to achieve this parity with ordinary data member access expressions.

2.6.1. Rvalue accessor conventions

When rvalue accessor overloads are provided, there is widespread disagreement about whether to return by value or by rvalue reference. Values are safer; references are faster. The standard library offers little guidance at present, but `std::get(std::tuple&&)` offers one precedent for return by reference, and `std::experimental::optional::get()` is another.

2.7. Diagnosis of dangling references

Beyond warning on temporary expressions bound to reference return values, compilers generally do not look for dangling reference problems. Functions that return xvalues are assumed to be safe. Not only is this a pitfall for beginners, it is a source of subtle problems in templates where a subexpression expected to be an xvalue turns out to be a prvalue, or a full-expression expected to be a lifetime-extended prvalue turns out to be an xvalue with a prvalue subexpression.

2.7.1. Hard diagnosis in NSDMIs

The recent resolution to CWG DR 1696 and 1815 forbids temporary expressions from nonstatic data member initializers and constructor mem-initializers ([class.base.init] §12.6.2/8, ¶11), because they would create dangling references. This solution does not cover aggregate members containing references nor `initializer_list`s; instead they gain undefined behavior because the resolution removes the text that would specify their lifetimes. Clang, the only current implementation of these resolutions, diagnoses all the cases equally. In this case, it appears that the language specification is lagging behind the implementation. Optimal diagnosis is based not on a particular initializer being a temporary expression, but on the actual application of lifetime extension which would certainly create a dangling reference. The current standard (with or without N3918) lacks conceptual terms to reason about this.

3. Analysis

Dangling references may be remedied by applying lifetime extension to all the temporary objects that are reachable from the result of a full-expression. The candidate objects are those representing prvalue subexpressions. The nodes in the reachability graph are the results of all the expressions and conversions. The edges are determined by the operators, according to simple rules.

The following subsections will consider properties of the various operators. These are distilled into a general rule in the following *Proposal* section.

3.1. Referents of glvalues

Aside from `typeid`, which has no reach, the built-in operators and conversions which yield glvalues follow a consistent pattern:

- Any glvalue expression has at most one glvalue operand, after its required lvalue-to-rvalue conversions, and considering discarded operands as cast to `void`. If there is such a glvalue operand, the expression refers to the same complete object.
  - The one possible exception is an access (dot operator) to a nonstatic member with reference type. However, if the initializer of the member reference involves `this`, it may refer into the object expression (the left-hand side). It is safest to assume it does, and treat this case uniformly as if the member reference was a member object.
- An assignment expression additionally has a prvalue on the right-hand side. This is not reachable unless it has pointer or class type. These cases are considered in the next subsections. A pointer-to-member access also has an unreachable prvalue right-hand-side.
- A glvalue with only a prvalue operand is either a cast which refers to its operand, or a pointer indirection (unary `*`) which refers to the referent of its operand.
- A nullary glvalue expression is an id-expression and may be assumed not to refer to a prvalue.
- Operators whose behaviors are defined in terms of other operators, such as compound assignment and subscripts, are treated in terms of the more fundamental operations.

In summary, if an expression is a glvalue and it has at least one operand, the first operand is reachable.

3.2. Referents of pointers

The case of pointer indirection is useful because it describes the behavior of arrays. N3918 includes a special case for a subscript (`[]`) expression applied to an array prvalue, but rules similar to the above generalize to all built-in pointer expressions and conversions.

In this subsection, the terms “referent” and “refers to” relate to indirection of pointer values, not lvalues of pointer objects. *New-initializers* are always unreachable and will be ignored.

- Any prvalue or assignment expression of pointer type has at most one operand which is a pointer prvalue, after any required array-to-pointer or lvalue-to-rvalue conversion, and ignoring discarded operands. If this operand exists, the result refers to the same complete object, and any other operand is an integer prvalue.
- The operand of an array-to-pointer conversion is reachable from its result.
- A prvalue expression of pointer type, with a sole operand that is not a pointer prvalue or null pointer constant, is either a `reinterpret_cast` applied to an integer prvalue, which may be considered to refer to nothing, or a unary `&` expression applied to an lvalue, to which it refers.

- A nullary prvalue pointer expression is `nullptr` or `this` and may be assumed to have no reach.
- A glvalue of pointer type with an operand of glvalue pointer type refers to the same object, hence it has the same referent, unless it is an assignment expression.
  - The referent of a pointer assignment expression is the referent of the right-hand side. Only by extreme convolution may prvalues appear in the left-hand side of a pointer assignment expression, so it is reasonable to consider them reachable by lifetime extension, by default.
- Operators whose behaviors are defined in terms of other operators are treated in terms of the more fundamental operations, as before.

Generalizing from these observations, an operand of a pointer-typed expression is either a reachable lvalue, an unreachable pointer with a reachable referent, or an unreachable integer prvalue. The array-to-pointer conversion also reaches its source.

To simplify this further, note that prvalues of integer and pointer type need not be backed by objects in the first place. If a backing object does exist, it is trivially destructible so its lifetime may be ignored. So, consider them as reachable, with any resulting lifetime extension nullified by the as-if rule. Then, an operand of a pointer expression is either a reachable lvalue, a reachable pointer with a reachable referent, or a reachable integer prvalue. In short, any operand of a built-in pointer-type expression or conversion is reachable.

3.3. Reachability from class prvalues

The only built-in operations that produce prvalues of class type are aggregate initialization and trivially-copyable lvalue-to-rvalue conversion. Operator overloading and initialization by constructor are handled as function calls; see §3.4 below.

3.3.1. Aggregate-initialized temporaries

An explicit type conversion expression (e.g. `T{x, y}`, [expr.type.conv] §5.2.3) implemented by aggregate initialization ([dcl.init.aggr] §8.5.1) reaches the initializers of members of reference, pointer, or class type. The existing rule for named declarations is simply extended to temporaries.

Brace-or-equal initializers of member declarations (a.k.a. NSDMIs) evade elegant semantics. See CWG DR 1696 and 1815. Ideally NSDMIs would be eligible for lifetime extension when participating in aggregate initialization, but that would be inconsistent with their treatment by constructors. This proposal follows the DR 1696 recommendation to forbid NSDMIs from binding a prvalue to a reference, regardless of whether the class is an aggregate. This is extended to forbid implying any lifetime extension, not limited to initializers of reference members.

Note that a pointer prvalue may refer to a temporary, and it may also be backed by a temporary object in its own right.

```
/*   Create an int temporary initialized with 5, place its address in an int const* temporary, and
     preserve both temporaries.  */
int const * const & p = & static_cast< int const & >( 5 );
```

In this respect, a pointer behaves like a class with a nonstatic member reference.

```
/*   Create an int temporary initialized with 5, bind it to the int const& member of a class-type
     temporary, and preserve both temporaries.  */
struct { int const & r; } const & p = { 5 };
```

3.3.2. Prvalue copies

An lvalue-to-rvalue conversion result of trivially-copyable class type (unions included) reaches its glvalue source if the class contains any nonstatic member of a type that bears a referent, such as a reference or a pointer. This quality is recorded by marking its implicit copy and move constructors with `export`; see below. In turn, any class may be considered to bear a reference on the basis of this `export` decoration. (This rule applies by default to non-trivial classes as well.)

```
struct tci { int i; }                  // tci holds an int and it is purely value-semantic.
struct tcp : tci {              // tcp holds an int and an int*, adding reference semantics.
    int * p = & i;
};
struct d : tcp {    // d ensures that any reference is a self-reference, restoring value semantics…
    d( int in ) { i = in; }
    d( d const & in ) { i = in.i; }    // … which requires defining a copy constructor.
};
tci a = d{1};                                        // Lifetime of d{1} is not extended.
tcp b = d{2};                                 // Lifetime of d{2} is extended and b refers into it.
d c = d{3};                                          // Lifetime of d{3} is not extended.
```

3.3.3. Prvalues used as values

Lifetime extension of class objects may be expensive, so it should be avoided where unnecessary, when the class object is used only for its value and its identity is inconsequential. Any constructor can save the value of `this`, but a copy or move constructor should provide for copy elision when its argument is a temporary. Behavioral dependence on copy elision renders a program unportable. Therefore, it is best to exclude copy/move elision candidates ([class.copy] §12.8/31) from lifetime extension, while otherwise performing normal reachability analysis and lifetime extension.

```
struct a { int const & r; } obj = a{ 5 };          // 5 is extended; a{…} is not.
```

In this example, `a{ 5 }` and `5` are both reachable prvalue subexpressions, but only `5` receives lifetime extension because copy elision strips `a{ 5 }` of its object-ness.

This rule does not need to be restricted to classes. As in the previous section, the same principle applies to pointers. (As noted in §3.2 above, the lifetime of an unreachable POD object is unobservable, so the as-if rule could strip such a lifetime extension anyway.)

3.4. User-defined lifetime extension

User-visible hooks are needed to define reachability for function calls, overloaded operator expressions, and initialization by constructor call. The result of such an operation reaches argument expressions that initialize specially designated (i.e. `export`) parameters. The implicit

`this` parameter of a constructor is identical to its result; there is no such thing as undoing its reachability.

3.4.1. Implicitly-declared member functions

As mentioned in §3.3.2, a trivially-copyable class is determined to bear a reference if it contains a member that bears a reference, and if so, its trivial member functions are marked with `export`. This rule extends to nontrivial classes by computing each `export` designation as the sum (maximum) of the `export` designations of the corresponding subobject operations, much as `noexcept` specifications are computed.

3.4.2. Perfect forwarding

C++ encourages passing value-semantic parameters by reference, for the sake of efficiency. Unfortunately, this muddles the distinction between values and references. A reference parameter bound to a prvalue argument should behave like pass-by-value when the parameter is used *within* the function as a value (per §3.3.3 above): The argument should propagate reachability analysis, but be individually exempt from lifetime extension.

At least one solution has been proposed, in N3538 *Pass by Const Reference or Value* (Crowl, rev. 2, 2013). The idea is to add a declarator *ptr-operator* with `|` instead of `&`, expressing that the user does not care whether a parameter really is a reference. However, this only attacks the low-level problem of optimality across ABIs. The natural extension to perfect forwarding (presumably spelled `||`) was not explored, at least in any registered document, probably because modifiable access with ambiguous aliasing would harm interoperability — although even for `|`, `mutable` already cancels `const`.

Considering the alternative as a dead end, a slightly modified "by value" `export[=]` specifier is proposed, which is intended to resemble a by-value lambda capture, or evaluation in general. The effect is to exempt any prvalue bound directly to the parameter from lifetime extension.

Association by value is considered as a lesser degree of association, as it enables lifetime extension for a subset of the eligible objects. A parameter of an implicit special member function may be designated `export[=]` if the class has a nonstatic member with a matching `export[=]` member function. This is the default qualification for classes with member pointers and references, as they are copied by value. Full `export`-level qualification of a special member function can only originate from an actual specifier. It would carry the semantic meaning of a class containing a self-reference, which continues to refer to the old object after a copy/move — an impossible design pattern.

4. Proposal

Within the initializer of an object or reference with static, thread, or automatic storage duration, certain prvalues are represented by objects with the same storage duration as the declared entity. These prvalues are identified by the relationship of *lifetime association* with the enclosing full-

expression. Prvalues in other contexts or lacking this relationship, or which are eligible for copy/move elision ([class.copy] §12.8/31), are represented when necessary by temporary objects.[4]

The result of an expression not involving a function call ([expr] §5) is lifetime-associated with an operand value, and the result of a conversion ([conv] §4, [over.best.ics] §13.3.3.1) is associated with its source value, if the result is a glvalue of object type or if it has class or object pointer type ([basic.compound] §3.9.2/3). Expression operands are associated after any conversions required by the expression, including those in [expr] §5/9-10 and list-initialization conversions ([over.ics.list] §13.3.3.1.5), and treating discarded-value expressions ([expr] §5/11) and any object expression in a static class member access expression ([expr.ref] §5.2.5/4.1) as prvalue expressions of type `void`. Lifetime association is transitive, so an expression result may be indirectly associated with any subexpression. However, no association applies to an operand of a `typeid` expression or to a *new-initializer*.

4.1. Function types

In addition to a return type and a parameter type list, a function type also has one *lifetime qualifier* per parameter, including the implicit object parameter for member functions ([over.match.funcs] §13.3.1/3). Each qualifier is either *lifetime-unqualified*, *lifetime-qualified by value*, or *fully lifetime-qualified*.

The result of a function call, or an initialization by a constructor, is lifetime-associated with the values that initialize its lifetime-qualified parameters. Arguments passed by value (or to a by-value qualified parameter) are ineligible for lifetime extension, although they may carry lifetime-association to their subexpressions.

```
template< typename ret, typename arg >
ret fn( export arg );

// string temporary "A" is extended. Presumed meaning: fn returns a reference to its parameter.
std::string && r = fn< std::string &&, std::string && >( "A" );

// Temporary "B" is not extended because string s move constructor doesn't propagate association.
std::string s = fn< std::string &&, std::string && >( "B" );

// Temporary "C" is not extended because it is passed by value. The reference must come from elsewhere.
std::string && t = fn< std::string &&, std::string >( "C" );

// Temporary "D" is extended. The return value of fn is as well.
std::string && u = fn< std::string, std::string && >( "D" );
/*  In this last case, fn is a factory function. Like a constructor, it has the prerogative to associate
    arguments to its result. With a return value class like string, with well-defined copy and move
    constructors that do not propagate association, associating arguments is fairly senseless.  */
```

Function types are partially ordered with respect to lifetime qualification. One function type is more lifetime-qualified than another if each of its lifetime qualifications is stricter, provided that

[4] With these constraints, the term "temporary object" may easily be replaced by a storage duration. See also CWG DR 1634.

both lifetime-unqualified types are the same. An object of pointer to function type or pointer to member function type may be implicitly converted to a more lifetime-qualified type. An explicit `static_cast` may convert such a value to a type with different lifetime-qualification. Neither of these conversions may change qualifications that are not immediate to the type, such as within parameters of pointer-to-function parameters or within pointers to pointers to functions. (Thus, a caller may explicitly grant or refuse a scoped lifetime to a passed-by-reference prvalue object, by casting a function before calling it. However, a caller may not rewrite lifetime requirements that are intrinsic to interfaces used by other parts of the program.) Reference-to-function type conversions correspond to pointer type conversions as usual.

```
int&& (*f)( export int&& )                  // OK, implicit conversion adds qualification.
    = []( int&& i ) -> int&& { return std:move( i ); };

int&& (*bad)( int&& ) = f;                  // Error, cannot implicitly remove qualification.
int&& (*ok)( int&& )                        // OK, explicit conversion removes qualification.
    = static_cast< int&& (*)( int&& ) >( f );

int&& (**ppf)( export int&& )                  // Error, cannot add nested qualifications.
    = static_cast< int&& (**)( export int&& ) >( & ok );
```

4.2. Function declarations

parameters-and-qualifiers:
`(` *parameter-declaration-clause* `)` *accessor-specifier*$_{opt}$ *cv-qualifier-seq*$_{opt}$
ref-qualifier$_{opt}$ *exception-specification*$_{opt}$ *attribute-specifier-seq*$_{opt}$

lambda-declarator:
`(` *parameter-declaration-clause* `)` *lifetime-specifier*$_{opt}$ `mutable`$_{opt}$
exception-specification$_{opt}$ *attribute-specifier-seq*$_{opt}$ *trailing-return-type*$_{opt}$

parameter-declaration:
attribute-specifier-seq$_{opt}$ *lifetime-specifier*$_{opt}$ *decl-specifier-seq declarator*
(etc. for other *parameter-declaration* production rules)

accessor-specifier:
`export [ const ]` — *const-covariant synonym for* `export[&]`
lifetime-specifier

lifetime-specifier:
`export` — *synonym for* `export[&]`
`export [ ]` — *explicit non-qualification*
`export [ = ]` — *associate result to parameter value*
`export [ & ]` — *associate result to parameter object*
`export [ auto ]` — *compute association from return*
`export [` *identifier* `]` — *transfer association from another signature*

A parameter is fully lifetime-qualified if it is declared with an `export[&]`, `export`, or `export [const]` specifier. It is lifetime-qualified by value if declared with the `export[=]` specifier. One may be explicitly declared as lifetime-unqualified by the `export[]` specifier. An *accessor-specifier* or *lifetime-specifier* following a *parameter-declaration-clause* qualifies an

implicit object parameter. Only named nonstatic member functions (excluding constructors and destructors) may have *accessor-specifiers*.

All declarations of a function shall specify identical lifetime qualification. No diagnostic is required for this rule (particularly between different TUs).

A virtual override may be less lifetime-qualified than the overridden function, but not otherwise differently qualified. The qualifiers of the function statically chosen by overload resolution are used for a given call.

For implicitly-declared special member functions, an `export` specifier is applied to a function parameter if the result of the corresponding operation on any of the class members (copy/move construction/assignment) would be lifetime-associated with its source. The *accessor-specifier* of an implicitly-declared assignment operator is `export`. As with `noexcept` specifications, these specifiers are also implied when a member is defaulted on its first declaration. Unlike `noexcept`, an explicitly-specified lifetime-specifier or accessor-specifier may differ from its default. (Disabling a default `export` is accomplished by `export[]`.)

```
struct s {
    int & r;
    s( export int & in ) : r( in ) {}

    s( s const & in ) = default;             // in is implicitly fully lifetime-qualified
    s( export s && in ) = default;           // in is explicitly fully lifetime-qualified
    // in is explicitly not lifetime-qualified. The implicit object parameter is implicitly fully qualified:
    s & operator = ( export[] s const & in ) = default;
    // Neither in nor the implicit object parameter are implicitly qualified:
    s & operator = ( s && in );
};
```

4.2.1. Automatic qualification

The remaining two forms provide calculated lifetime qualification. The `export[auto]` specifier provides automatic computation. It inspects all the `return` statements in the function, and applies to its parameter the strongest lifetime association found between a return value and an expression naming that parameter. Such an expression may be an *id-expression*, `this`, or (perhaps implicitly, [class.mfct.non-static] §9.3.1/3) `*this`. The expression `*this` is considered atomically: When it initializes a lifetime-qualified by value parameter (such as when it is used as the source of a copy-constructed value), the lifetime association of the implicit object parameter is by value. Otherwise, a lifetime-associated `this` implies full lifetime qualification.

```
struct bound {
    int bound_arg;

    int plus_by( int i ) export[auto]    // export[auto] resolves to export[].
        { return bound_arg + i; }
    int & inc_by( int i ) export[auto] // export[auto] resolves to export[&].
        { return bound_arg += i; }

    template< typename ptmf, typename ... param >
```

```
    decltype(auto)
    apply( ptmf fun, export[auto] param && p ) export[auto] {
        return (this->*fun)(bound_arg, std::forward< param >(p) ...);
    }
};

int const & a = bound{ 1 }.plus_by( 1 );  // 1. No lifetime extension for bound{1}.
int const & b = bound{ 2 }.inc_by( 2 );     // 2. Lifetime extension for bound{2}.
int const & c = bound{ 1 }.apply( & bound::plus_by );         // Same as 1.
int const & d = bound{ 2 }.apply( & bound::inc_by );          // Same as 2.

struct haz_ptr {
    int * ptr;       // Pointer member causes move constructor parameter to be export[=].

    // Returning by value causes export[auto] to deduce export[=] from return *this.
    haz_ptr operator ++ (int) export[auto] {
        struct inc { haz_ptr *self; ~ inc() { ++ self->ptr; } }
            guard{ this };// Use scope guard to change state after constructing return value.
        return * this;
    }
    // Returning by reference causes export[&] to be deduced from return *this.
    haz_ptr & operator ++ () export[auto] {
        ++ ptr;
        return * this;
    }
};
```

A function declarator containing the `export[auto]` specifier denotes a placeholder function type. As with return type deduction, the function type shall not be used before it is deduced. Deduction occurs at the end of the *function-body*, so such functions cannot recurse. There shall be at least one `return` statement. A constructor declaration shall not use this specifier.

4.2.2. Reflection

The `export[`*identifier*`]` specifier provides for reflective abstract manipulation. When used in a deduced context in a template declaration, the identifier is declared in a special name space of lifetimes. As with labels, this environment is separate from and invisible to ordinary name lookup, and the visibility of such a declaration extends backward to preceding parameter-declarations. The declared name is bound to the corresponding qualification from the function type. When used elsewhere, it applies the extracted qualification. (Note that such manipulations should seldom be necessary. They are firmly in the domain of reflective metaprogramming.)

```
template< typename ret, typename ... param >
struct implicit_object_to_parameter
    // Declare names for lifetime qualifications of a function type as it is decomposed.
    < ret( export[ param_life ] param ... ) export[ this_life ] > {
    typedef ret ( * type )
        // Reassemble a new lifetime-qualified function type.
        ( export[ this_life ] obj *, export[ param_life ] arg ... );
};
```

4.3. Covariant accessors

A lifetime-qualified implicit object parameter indicates that a member function's return value references data owned by its object. If the object is expiring, then the data certainly is as well. The same property applies, as the user specifies, to non-modifiability via `const`. The reference types of the implicit object argument and the return value are covariant because they conceptually refer to parts of the same abstract value. This is also the idea expressed by `export`, so `export` implements such covariance.

4.3.1. Rvalue accessors

When an *accessor-specifier* appears with an lvalue (`&`) ref-qualifier, and the return type is declared with the `&` qualifier, an additional *rvalue accessor* overload may be implicitly declared with both `&` punctuators transformed to `&&`. The specific preconditions are:

1. The declarator `D1` contained in the function declarator as per [dcl.fct] is a *declarator-id*, i.e. the declaration is of a member function and not something like a pointer to a function.
2. The *ref-qualifier* of the function declarator is `&`.
3. No member function is explicitly declared in the class with a *ref-qualifier* of `&&`, and the same name, parameter list, and *cv-qualifier-seq*.
4. The function declarator has an *accessor-specifier*.
5. According to [dcl.ref], the function declarator appears as `D1` in a declarator of the form
   `&` *attribute-specifier-seq*$_{opt}$ `D1`
   i.e. the function return type is immediately declared as a reference.
6. If the reference declarator from the previous rule is the complete declarator, the *decl-specifier-seq* of the declaration specifies a non-reference type or a dependent type [5].

If the implicit object parameter of the implicitly-declared function is lifetime-qualified, the return type is transformed by modifying the `&` punctuator identified in precondition #5 into `&&`. Otherwise, the return type is unmodified. For templates, this must occur during specialization of the signature.

If an implicitly-declared rvalue accessor is a virtual function, the program is ill-formed.

A specialization of an implicitly-declared rvalue accessor template participates in overload resolution only if it satisfies precondition #6, which forbids reference collapsing.

The user may also declare an rvalue accessor as explicitly defaulted. This is allowed even if the preconditions relating to the return type (#5-6) are unmet for the original function. Conversely, "`= default`" is allowed for any function declaration mentioning "`&& export`" given a corresponding "`& export`" overload. The return type, *lifetime-specifiers*, *accessor-specifier*, and

---

[5] 1) Throughout this proposal, assume that any *trailing-return-type* is substituted for its *decl-specifier-seq*. 2) The *decl-specifier-seq* may indicate a reference type `R` yet provide for a covariant rvalue accessor only if the function returns a reference to pointer to [member] function returning reference type, e.g. `R (*& fn () export &) ()`. 3) Function types are not reference types, but formation of an rvalue reference to function type will result in an ill-formed program or template substitution failure.

exception-specification need not match that of the preempted implicit declaration. Such a template specialization may be called even if the *decl-specifier-seq* specifies a reference type. (Note, if the return type is as-implicit and the dependent type specifies an rvalue reference, the function will propagate rvalue category from the object expression.)

Definitions of rvalue accessor functions follow the rules of special member functions: an implicit definition is provided when one is ODR-used or explicitly defaulted after its first declaration. An implicitly-defined rvalue accessor acts as a forwarding call wrapper to the original lvalue-qualified function. If the implicit object parameter of the original function is lifetime-qualified, and the return type of the rvalue accessor is not an lvalue reference type, the result is treated as an xvalue. Otherwise, the result initializes the return value normally.

4.3.2. Const accessors

The `export [const]` *accessor-specifier* indicates to use a common function body for `const` and non-`const` accessor overloads, and to generate a covariant return type. It is ill-formed unless:

1. The declarator `D1` contained in the function declarator as per [dcl.fct] is a *declarator-id*, i.e. `export [const]` can only decorate a member function.
2. The *cv-qualifier-seq* of the declarator does not include `const`.
3. No member function is explicitly declared in the class with the same *ref-qualifier*, name, and parameter list, but with an added `const` *cv-qualifier*.
4. The function return type is declared with a placeholder type or it is a reference or pointer type. If the *decl-specifier-seq* is dependent, this constraint applies upon specialization.

A function declaration with `export [const]` is treated as two non-implicit declarations (or definitions) differing in that one includes `const` in its *cv-qualifier-seq*, and one does not. Both use the `export` lifetime specifier. The return type of the non-`const` function is as declared, and the declared return type of the `const` overload is modified as follows:

1. If the declarator immediately enclosing the function declarator is a pointer or reference declarator, its modified type `T` is replaced with `T const`. This has no effect if `T` is a reference type.
2. Otherwise, if the return type is a pointer type `T*` or reference type `T&` or `T&&`, its referent `T` is replaced with `T const`.
3. Otherwise, by logical deduction, the declared return type must be `decltype(auto)` or `auto`, and it is not modified.

There is no requirement that the unmodified type `T` lack `const` qualification. If it is already qualified, there is no covariance.

A function declaration generated by `export [const]` may cause the implicit declaration of an rvalue accessor, or may explicitly declare an rvalue accessor.

A function definition declared `export` or `export const` may match an `export [const]` declaration, but an `export [const]` definition can only match an `export [const]` declaration.

4.4. Name mangling

The `export` specifier modifies function types as does the strongly-typed `noexcept` function specifier, so this proposal affects ABIs analogously to N4533.

Function overloads cannot differ only in lifetime qualification, so lifetime qualifiers need not be included in mangled function names. However, lifetime-qualified function types may appear in parameter lists and template argument lists, which do require mangling.

Mangled names will change when they include template arguments containing the types of standard library functions, for example the return types of specializations of `std::mem_fn` over `std::vector::back` or `std::bind` over `std::get`, as qualification is added to such standard library functions. However, such things seldom occur in binary library interfaces. Observation of the types of standard library member functions is never kosher.

4.5. Library support

Reference-semantic standard getter functions such as `std::get` and `std::array::front` are lifetime-qualified. Enumeration of all cases is beyond the scope of this proposal.

Forwarding call wrappers, such as the call operators of `std::reference_wrapper` and the bind expression class, may uniformly apply `export[auto]` for all lifetime-specifiers and the accessor-specifier. (A `reference_wrapper` does not need an accessor-specifier, though.)

Arguments which are retained in a wrapper, such as by functions `std::bind` or `std::async`, or classes `std::function` or `std::tuple`, should be lifetime-qualified by `export[auto]` as well.

4.5.1. `std::function`

`std::function` must be specialized over signatures with lifetime qualifiers. (For such qualifications in general, see P0045 §2.) The given signature applies to its member call operator. The behavior differs only at call sites that observe lifetime extension, so type-erased dispatch islands need not be parameterized over lifetime qualification. The signature may apply a lifetime qualifier to the implicit object parameter, indicating that the return value may reference the type-erased functor object.

The generic conversion constructor of `std::function` is declared `explicit` if the encapsulated function call would be lifetime-associated with a parameter that does not have lifetime qualification. Thus, implicit and explicit conversions between `std::function` types work analogously to conversions between function types.

5. Usage

Lifetime qualification describes dependence between different variables. It need not be used for functions and class interfaces involving only self-contained values. Otherwise, `export[auto]` should deduce the right thing when each `return` statement mentions all its input parameters. Apply `export[auto]` to any reference, pointer, or similar parameter that can reach the return value. Mention it with perfect forwarding when the return value could refer somewhere.

This proposal does not demand immediate adoption everywhere. It is a non-breaking change, and opting-in to the feature will be done on a case by case basis. Generic and foundational facilities should be updated first, and see the highest impact. More ordinary, value-heavy classes should care less, aside from reference-semantic accessors which this proposal sugars.

Failing the easy cases, and given the motivation to apply annotation, the "hard" usage is:

5.1. `export`

If destroying a given parameter after the function return would invalidate the result value, then mark that parameter as `export`.

This covers the most familiar cases: parameters that are accessed by an accessor function, references retained by a constructor, returning `*this` for method chaining, a container parameter that yields an iterator, range, or view, etc.

It is harmless to apply `export` to accessors that return by value. For a template member function that could specialize to return a pointer (or pointer-like) value that ultimately reaches a live temporary, it is better to `export` the owning parameter.

5.2. `export[=]`

If the result type is reference-semantic (it refers to or reaches some non-owned object), and the result and a given parameter may both reach object(s) from the same owner, then mark that parameter as `export[=]`.

This covers most operations on iterators, smart pointers, query classes, and similar objects used for navigating data structures. The term "referent" and "identified by" are used loosely. The purpose of this sort of association is to connect a chain of navigation operations back to an accessor method of the actual owning container.

`export[=]` may be considered as advanced usage. In the worst case, using `export` instead will only cause some pointer-like objects to live longer, and such objects tend to be small and passive. (Owning and shared-ownership smart pointers are not passive, but implementing such things is advanced usage.)

6. Polymorphic lifetime analysis

Lifetime extension is a feature unique to C++, to the author's knowledge. Static analysis of variable lifetimes for diagnostic purposes has been studied in full-scale systems since the mid-1990's. This deeper analysis attaches annotations to object types, introducing a sort of polymorphism where compatibility of annotated types is determined by annotation "subtyping."

The subject of such analysis is to determine that no reference outlives its referent. In addition to basic safety, ideally it extends the reach of ordinary programmers, to write well-disciplined code leveraging scoped memory pools or "regions." Region-based memory management offers the best of garbage collection and heap allocation.

Initial success was accomplished by extending the functional language ML. The goal was to introduce regions to reduce reliance on mark-sweep garbage collection.[6] Later, the C-derived Cyclone language applied the same technique to improve the safety, determinism, and performance of heap allocation.[7] Recently, the Rust language was established specifically as a competitor to C++, with lifetime annotations and analysis in the initial feature set. However, it is still early days for the wider software development community, and best practices in Rust are still being developed.

In the past year, Microsoft has been developing similar lifetime diagnostics in MSVC. The compiler determines most annotations automatically, which the user may amend with the `[[lifetime( other_variable )]]` attribute.

The current proposal extends to encompass rich lifetime annotations, following the attribute initiative. Whereas lifetime-association is currently defined as a relationship between a result and its operands, it may also apply between operands alone.

lifetime-specifier:
`export`
`export (` *lifetime-association-seq* `)`
`export ( auto )`

lifetime-association-seq:
lifetime-association
lifetime-association-seq `,` *lifetime-association*

lifetime-association:
lifetime-id `=` `&`$_{opt}$

lifetime-id:
identifier
`this`
`return`

In this schema, the optional `(=)` indicating evaluation is extended to allow a sequence of parameter names. Each parameter is potentially "assigned from" any other parameter, with `this` standing for the implicit object. To avoid awkwardness, `return` explicitly nominates the return value for association, whereas currently it is the only alternative. An explicit `&` punctuator specifies that an address is used, which is currently the default. The *lifetime-id: identifier* production preserves the existing *lifetime-specifier:* `export[`*lifetime-id*`]` production. When used in a deduced context for reflection, it likewise allows pattern-matching on more sophisticated lifetime associations.

```
/*   Take a reference-semantic object, a first list of objects, and a second list of pointers to objects.
     Store the value of the object in the first list and the address of the object in the second list.
     Static analysis will guarantee that the object's referents life as long as the first list, and the object
     itself lives as long as the second list.  */
void add( export[ objlist =, ptrlist = & ] foo & obj,
```

[6] Tofte and Talpin. Region-Based Memory Management. *Information and Computation,* 132 (1997).

[7] Grossman et. al. Region-Based Memory Management in Cyclone. *ACM PLDI,* 2002.

```
        std::list< foo > & objlist, std::list< foo * > & ptrlist ) {
    objlist.push_back( obj );
    ptrlist.push_back( & obj );
}

/*  Take a function returning void with at least one parameter. Return a call wrapper function which
    returns the first argument.  */
template< typename p0, typename ... p >
auto return_arg1_fn( void (*f)( export[ L0 ] p0, export[ L ] p ... ) )

    // Preserve preexisting lifetime specifiers, and add an association from first param to return value:
    -> p0 && (*)( export[ L0, return = & ] p0, export[ L ] p ... ) {

    static auto fglob = f; // For illustration ;v) (This global deserves a lifetime annotation!)

    return + []( p0 a0, p ... a ) {          // Lifetimes could be annotated here…
        fglob( a0, std::forward< p >( a ) ... ) );
        return std::forward< p0 >( a0 );
    };                  // … but the implicit conversion on function pointers works just as well.
}
```

Lifetimes should remain ordered, despite this extension, so function-type conversions (§4.1 above) will continue to work. Ordering is a cornerstone of the overall analysis scheme.

7. Conclusion

Temporary lifetime extension has long provided a uniquely proactive solution using shallow, opportunistic lifetime analysis. It has considerable untapped potential. This proposal attempts to describe an incremental step in code expressiveness and compiler insight. It is yet unimplemented, and it was heavily revised and extended at the last minute to account for the possibility of deeper lifetime analysis, which was revealed days before the submission deadline. This is essentially a preview and a strawman intended to encourage review and to guide prototype development.

The benefits include

- fewer surprising dangling references (it bears repeating),
- better dangling-reference error diagnosis,
- better adaptor templates that need not perform complete encapsulation,
- improved language uniformity, and
- more diverse scope-based facilities in the future of the C++ language and its libraries.

7.1. Acknowledgements

Richard Smith provided helpful review and insight, especially on inroads in this issue by N3918 and lifetime qualification of non-reference parameters. He previously explored the design space.

At EWG review in Urbana 2014, Chandler Carruth raised the issues of generic and dynamic call wrappers, namely `std::function`. Ville and others raised virtual functions. Gaby suggested that exception specifications are a poor model. These concerns pointed the way forward.

Revision history

N4221, 2014-10-10: Initial revision.

P0066, 2015-09-28: Use strong typing instead of analogy to weak exception-specifications.
Move the `export` keyword after function parameter attributes.
Add `export[auto]` and `export[`*identifier*`]` for reflection.
Add `export[const]` and covariant accessors for convenience.
Add `export[=]` for efficiency.
Remove qualified type restrictions and decay rules of lifetime specifiers.
Simplify lifetime association rules.
Outline essential library support.
Rewrite and expand text.