Document number: N4221
Date: 2014–10–10
Reply to: David Krauss
(david_work at me dot com)

Generalized lifetime extension

1. Abstract

A model is introduced to describe the association between a value, and the object or objects needed to represent it. This enables a generalization of lifetime extension, applicable to constructors and accessor functions which opt in using the `export` keyword. The current lifetime extension behavior is also more elegantly described, and the "temporary expression" controversy is resolved. This resolves EWG 120 and many other dangling reference issues.

2. Background and motivation

Lifetime extension applies when a reference is bound to a temporary object or its subobject.

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

int main() {
    auto && whole = d{};
    b && base = d{};
    int && member = d{}.m;
    c aggregate = { d{} };
    std::cout << "wait for it\n";
} // Four d::~d() calls occur here.
```

2.1. Temporary expressions

It is impossible to apply lifetime extension to the result of just any function that might return a reference to a temporary. The general strategy in [class.temporary] §12.2/5 [1] requires statically resolving the referent object initializing a reference to see if it is temporary, but excludes objects to which something suspicious happened, such as binding to a function parameter. The problem is that dynamic behavior may be specified without using anything so suspicious.

```
struct self_ref { self_ref & me = * this; };
auto && q = self_ref{}.me; // No lifetime extension.
```

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

[1] References are to the C++14 FCD N3936 unless otherwise noted.

practice. Furthermore, disqualifications sometimes don't count, because evaluating the forbidden conditions also requires statically determining dynamic object identity.

```
struct self_ref_ctor {
    self_ref_ctor & me;
    // If *this is a temporary, it will be bound to a member
    // reference in a ctor-initializer, disabling persistence.
    self_ref_ctor() : me( * this ) {}
};
auto && q = self_ref{}; // In practice, extension still happens.
```

The current lifetime extension rules describe neither the intent nor the implementation practice. This is the gist of CWG DR 1299:

> The Standard is insufficiently precise in dealing with temporaries. It is not always clear when the term "temporary" is referring to an expression whose result is a prvalue and when it is referring to a temporary object.
>
> — *Johannes Schaub*

This issue has a resolution pending in N3918, but the last example remains inconsistent. N3918 defines "temporary" to apply to particular expressions that are known to refer to an object representing a prvalue expression within the same full-expression, but the bullet points of [class.temporary] still deal with requirements on the referent object. Considerable effort and finesse is needed to completely repair the original specification.

2.2. Naked member required

Lifetime extension does not apply to the result of a straightforward accessor function, so to obtain it through a nonstatic data member, that member must be the public interface.

```
class access {
    int m;
public:
    int & get_m() { return m; }
};

auto && i = access{}.get_m(); // No persistence, dangling ref.
```

(C++14 may actually require persistence here, as the object is never bound to a function parameter besides the implicit one. However, the intent of non-persistence is uncontroversial.)

Rvalue ref-qualified accessor functions are particularly hard hit, because it is known even within the function, and obvious to any well-informed user, that a reference return value will be dangling at the end of the full-expression. One solution is to return the owned object by move-constructed value, but this has not gained traction.

```
class has_vec {
    std::vector< int > m = { 1, 2, 3 };
public:
    std::vector< int > && get_ref() && { return std::move( m ); }
    std::vector< int > get_val() && { return std::move( m ); }
```

```
};
int main() {
    for ( int i : has_vec{}.get_ref() ) ; // Premature destruct.
    for ( int i : has_vec{}.get_val() ) ; // OK
}
```

Return by value only works in the second case if the value is self-contained. This is more restrictive than persistence of the containing object, as provided by binding to a member access expression.

2.3. Universal observation

Generalizing from the case of a member accessor, it is common to have a free function which returns some information within a given argument. The return value will remain valid after the full-expression if the argument is an xvalue resulting from applying `std::move` to an lvalue, but it will be a dangling reference if the argument is a prvalue. Such cases include the subject of EWG 120:

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

Here, Boost.Adaptors would allow only one adaptor to be applied, with the result being preserved by an internal variable using an `auto &&` "forwarding reference." When two adaptors are applied, the outer one survives but the inner one does not.

3. Analysis

The only objects actually eligible for lifetime extension are those representing prvalue expressions, which can be identified as targets within an enclosing initializer full-expression. If no prvalue were on hand, the implementation would have no suitable handle to a local object with variable lifetime. Allowing one full-expression to affect the meaning of another, without involving the type system, is generally unfeasible and undesirable.

In most cases, the prvalue corresponding to the referent of a declared reference may be statically determined: Simply start at the full-expression, and trace inward through each glvalue expression or conversion to an immediate operand or source value which refers to the same object. For example, the result of an assignment operator refers to its first operand, so if the full-expression is an assignment expression, that is its referent.

```
/*  Create a temporary initialized with 3, update it to 4, and
    extend its lifetime by binding it to a reference.
    Each cast refers to the same object as its operand. */
int & i = (const_cast<int&>(static_cast<int const&>( 3 )) = 4);
```

This might be neither readable, portable, nor implemented anywhere. However, it is a consistent interpretation of the longstanding rules, and the rules are reasonable except when they require

impossible analysis. N3918 takes a step in this direction by allowing the casts, but it does not allow the assignment. Such simple static analysis is low-hanging fruit, and it should be applied aggressively.

To apply lifetime extension to results of accessor functions requires a new language feature. Static analysis of function calls would also extend to user-defined conversions and operator overloads. Consistent, well-specified lifetime extension rules for standard conversions and built-in operators may provide the basis for such an extension and its usage.

3.1. Referents of glvalues

The built-in operators and conversions behave according to consistent rules, which may be leveraged to form rules for tracing from a given glvalue expression to the prvalue of its referent, without requiring any data storage besides what is already required for an AST:

- Any glvalue has at most one glvalue operand (or a prvalue operand of class type which is treated as an xvalue), 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 (or into) it.
  - The sole exception is a class member access to a nonstatic member reference. However, this is seldom applied to a temporary, and given a nontrivial constructor, the reference may well refer into the object expression anyway. It is reasonable to ignore the exception.
- A glvalue with only one 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 are treated in terms of the more fundamental operations.

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.

- Any prvalue or non-compound 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 considering discarded operands as above. If the operand exists, the result refers to the same complete object, and any other operand is an integer prvalue.
- A prvalue of pointer type with only one operand, which 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 not to have a prvalue referent.

- 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, which is covered by the first case.
- Operators whose behaviors are defined in terms of other operators are treated in terms of the more fundamental operations, as before.

A pointer prvalue may refer to another prvalue, and it may also be an 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 & p = & static_cast< int const & >( 5 );
```

In this respect, a pointer is not unlike a class with a nonstatic member reference.

3.3. Referents of classes

Aggregate-initialization of a member reference with a prvalue confers lifetime extension, because the lifetime of a member is the lifetime of the complete object ([basic.stc.inherit] §3.7.5). The user cannot currently achieve this with a nontrivial constructor. Emulating the aggregate initialization behavior using a constructor requires tagging constructor parameters, and then granting lifetime extension to argument expressions as appropriate.

For a class object to have a referent means that it somehow retains the address of another object. How (or whether) this is done depends on the user. The language must guarantee that if such a class prvalue is initialized by constructor, and its object has lifetime extension, then lifetime extension applies to arguments of tagged parameters as well. This is the same treatment given to the result and the operand of the unary `&` operator, except that there may be more operands.

Lifetime extension of class objects may be expensive, so it should be avoided where unnecessary, when the object is used only for its value and its identity is inconsequential. A constructor may always observe its own object's identity and record it in some persistent storage, but as a heuristic, a copy or move constructor should always allow for the possibility that its argument is a temporary. Behavioral dependence on copy elision renders a program unportable. Therefore, it is best to exclude copy elision candidates from lifetime extension, even while allowing it for the constructor arguments of a temporary which describe its value.

(As pointers are analogous to classes, a similar rule may be formulated. A pointer prvalue need not be lifetime extended, nor have a backing object at all, if it is ultimately used as a prvalue. However, this is already within the scope of the as-if rule, and likewise for any POD.)

3.4. User-defined lifetime extension

The above rules for built-in operators and class constructors naturally extrapolate to all functions. A function return value of reference type may refer to (or into) an argument, if the corresponding parameter is tagged. Any object backing a prvalue providing storage for the argument is eligible for lifetime extension. The same applies to a function return prvalue, which additionally is a lifetime extension candidate itself.

4. Proposal

Within the initializer of the variable or nonstatic member representing an object or reference with static, thread, or automatic storage duration, certain prvalues are represented by objects with the same storage duration. These prvalues are identified by the relationship of lifetime association with the enclosing full-expression, described below. Prvalues in other contexts or lacking this relationship are represented by temporary objects.[2] Initializers of variables granted storage duration by this rule likewise grant storage duration to their own inner prvalues. For this purpose, an init-capture is considered to initialize a subobject of the result of its lambda expression.

An expression result is lifetime-associated with an operand or argument 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] §5/8-9, and treating discarded-value expressions ([expr] §5/10) 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, these exceptions apply, in the given order and precedence, and not by transitive association:

- A conditional expression, a `typeid` expression, or a new-expression does not associate with its operands, including new-placement arguments.
- When an operand or source eligible for copy elision has a distinct object representation, it does not have a storage duration, although association proceeds normally.
- An expression or conversion implemented by a function call associates as determined by the function signature as described below.
- A compound assignment expression `A@=B` associates as if it were replaced with its functional equivalent `A=A@B`, where no lvalue-to-rvalue conversion is applied to the first `A`.
- An assignment expression of pointer type does not associate with the referent of its first operand. It associates with its first operand as an lvalue, and with its second, prvalue operand.

The result of a function call or an initialization by a constructor is lifetime-associated with the value that initializes a parameter, if the parameter declaration is preceded by the `export` keyword. The result of a nonstatic member function call is associated with its implied object argument ([over.match.funcs] §13.3.1/3) if it is declared using the `export` keyword after the parameter list. Such use of the export keyword and the potential association of a parameter's initializer are called *lifetime qualification*.

parameter-declaration:
`export`$_{opt}$ *attribute-specifier-seq*$_{opt}$ *decl-specifier-seq declarator*
(likewise for other production rules)

[2] With these constraints, the term "temporary object" may easily be replaced by a storage duration.

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

Lifetime qualification does not affect a declarator's type, and it may not be applied to a typedef-name. The association of semantics with the entity declared by the innermost enclosing declarator of function type ([dcl.fct] §8.3.5/3) is like that of exception-specifications ([except.spec] §15.4/2) except that there is no notion or requirement of compatibility.

A function type declarator may specify lifetime qualification only if its return value could have lifetime association, i.e. if it has class, object reference, object pointer, or template type-dependent return type. When the latter case resolves to an ineligible type, the qualifiers may be ignored. Any constructor may specify lifetime-qualified parameters. Constructors and destructors may not specify lifetime qualification of the implied object argument.

Likewise, a parameter-declaration may specify lifetime qualification only if it has class, object reference, object pointer, or type-dependent type.

Implicit move constructor and move assignment operator declarations provide lifetime-qualified parameters, but the corresponding copy operations do not. If a user creates a class containing non-owning references which are shared by copying and which potentially refer to objects in the scope of the class initialization, the user should explicitly declare the copy operations with lifetime-qualified parameters. In any case, lifetime qualification does not affect the validity of an explicitly-defaulted definition nor the triviality of the resulting function.

5. Rationale

In its longstanding form, lifetime extension prevents a dangling reference opportunistically, because the problem is preventable and obvious. The scope of this obviousness has expanded slightly over time, though. There is no reason not to maximize the remedy, and add persistence to all prvalue subexpressions backing would-be dangling references and pointers in the final result.

The primary rule and special cases of lifetime association are a condensed version of the general properties noted in the analysis (§3.1 and §3.2 of this proposal). These rules produce unneeded associations for some scalar prvalues, but they do not need object representation or lifetime extension according to the as-if rule. A good implementation should avoid overhead associated with any trivially destructible object that does not have a reference surviving the full-expression.

Prvalue expressions of non-pointer scalar type, which cannot refer to an object, do not confer lifetime extension to subexpressions that would otherwise be eligible. This minimizes the set of persistent objects, and helps to prevent abuse of the lifetime qualification facility for program semantics that do not involve return values. "Out parameters" are not supported as destinations for referents. Combination of this design pattern with persistent arguments would produce convoluted code. It would suggest that any subexpression could imply an object declaration. The storage duration of an out parameter is unknown and should not be presumed from the scope.

Not only does the new feature help prevent more dangling references, it exposes a form of object composition. The objects which receive lifetime extension from the initializer of a declarator can be considered to form an aggregate, together with the declared variable. This is similar to the

association between local variables in the same activation record, except that it can be encapsulated using functions. (To be clear, such aggregation has existed since C++98, but only with explicit use of aggregate initialization ([dcl.init.aggr] §8.5.1) as shown in the first example of this proposal.) Lifetime-qualification provides storage access to the calling function's scope that would otherwise require a carefully-designed template and the overhead of its instantiation.

The lifetime qualification syntax is designed to look like a storage-class-specifier. It indicates that the function may "export" the identity or properties of a parameter to its output. The effect is in fact a change in storage duration. It appears on the left-hand side of a parameter declaration to avoid appearing to associate with a default argument. Modification to the *noptr-declarator* and *noptr-abstract-declarator* productions is avoided because the associativity may be confusing, and several special cases would be required to provide similar usability consideration as attributes. When qualifying the implicit parameter, `export` appears after a parameter list, before any cv-qualifier-seq and ref-qualifier. This gives the impression of an elided decl-specifier-seq for the implicit parameter, between its "storage class" of `export` and its cv-qualifiers.

Declaring that an argument should survive the full-expression enables accurate diagnostics when lifetime extension cannot be applied, for example in a mem-initializer or new-initializer.

Conditional expressions cannot confer lifetime extension because one of the alternative operands will not exist. This issue was investigated in CWG DR 446 and the resolution considered lifetime extension, ameliorating the problem by making the result a prvalue (a new copy) if either operand is a prvalue. The value of the entire conditional expression will usually be self-contained. If it receives lifetime extension, and a subexpression of the second or third operand would be eligible but for the restriction, the implementation should produce a warning. This proposal has no particular incompatibility with persistent objects of conditional existence.

The formulation is designed to help resolve CWG DR 1299, 1634, and 1651.

6. Examples

Aggregate-like classes like `std::tuple` and `std::pair` may confer lifetime extension, even through getter functions. (The same applies to user-defined classes, which are usually more appropriate than such generic solutions.)

```
template< typename first_type, typename second_type >
pair< first_type, second_type >::pair
    ( export first_type const & in_first
    , export second_type const & in_second )
    : first( in_first ), second( in_second ) {}

template< size_t n, typename first_type, typename second_type >
tuple_element_t< n, pair< first_type, second_type > > &&
get( export pair< first_type, second_type > && ) noexcept;

template< typename value >
std::pair< value &&, sentry > bundle( export value && in )
    { return { std::forward< value >( in ), {} }; }
```

```
for ( int i : get< 0 >( bundle( std::vector< int >{ 1, 2 } ) ) )
    ; // The temporary vector remains valid.
```

Observer classes like `string_view` can externally retain a container, to avoid dangling in an easily-accessible case. *(Credit: Richard Smith.)*

```
string foo();
void f() {
  string_view v = foo(); // Lifetime-qualified conversion ctor.
  g(v); // v observes the object returned by foo().
}
```

Ordinary pointers and iterators provide the same safety.

```
namespace std {
string::iterator string::begin() export
    { return iterator{ this->start_ptr }; }

char & string::iterator::operator * () export
    { return * this->ptr; }
}

std::string::iterator it = std::string( "hello" ).begin();
char * s_ptr = & std::string( "there" )[0];

s_ptr[ 3 ] = it[ 2 ]; // OK: both std::strings still alive.
```

Where dangling pointer issues cannot be fixed, the extension is expressive enough to allow diagnosis.

```
// Warning: new-initializer contains expiring references.
auto x = new string_view( string( "hello" ) );

struct svc {
    string_view v = string( "there" ); // Similar warning.
};
```

Adaptor class templates intended for use as temporaries may be used as local variables, simply by adding the `export` qualifier as needed. From EWG 120:

```
std::vector<int> vec;
for (int val : vec | boost::adaptors::reversed
                   | boost::adaptors::uniqued) {
    // OK if operator | () exports its LHS.
}
```

7. Categorically qualified classes

Testing in conjunction with N4149 *Categorically qualified classes* is needed, if it is accepted. There are complex interactions as this proposal adds storage duration to anonymous objects, but that proposal forbids objects of certain classes, which behave like expression templates, from having storage duration.

7.1. Necessary support

An additional lifetime extension exemption is warranted, similar to copy elision candidates. Objects which are implicitly converted according to an rvalue (`&&`) class qualifier may receive lifetime extension only if lifetime qualification is applied and the conversion is to reference type, which together indicate that the return value is a subobject or property. This effectively violates the invariant of the qualifier, but it is according to the specification of the conversion. (Note that a converting constructor may be chosen instead of a conversion function to reference type, if the declarator is not of reference type. Persistence may depend on overload resolution, but unlike the case in the following section, it will not affect overload resolution.) As with copy elision candidates, if the selected conversion function or converting constructor is lifetime-qualified, the source object can still confer lifetime extension.

N4149 forbids binding a reference to a subobject of such a class, because if the complete object were converted to a form amenable to persistence, the result of conversion would not have the subobject. This concern applies to getter member functions as well: an object granted storage duration by its binding to an implicit object parameter cannot be replaced by something else. (Even if the resulting member access worked, it would not make sense.)

7.2. Overload-dependent lifetime

However, it should make sense to convert other arguments into persistent or temporary form. Such conversions may be considered by overload resolution. Candidate functions requiring forbidden lifetime extensions, or missing required lifetime extensions, may be removed from the set of alternatives, and replaced with functions which are viable for the result of class-qualification conversion. This conversion could be considered as a new, additional conversion sequence for an argument, applied before the usual implicit conversion sequence.

The existing sequence ranking process might work with the catenation of the two subsequences. N4149 avoids involving overload resolution for the sake of simplicity, and the conversions it requires do not affect rank. The absence of a class qualification conversion would seem to indicate a better-matching overload, though.

The product of such overload resolution should work well because the result of expression template evaluation is typically a self-contained object, defined by having no dependencies on the source temporaries. Conversion functions for rvalue qualification should have no lifetime qualification, thus eliminating the expression template as a lifetime extension candidate.

Such a solution would make expression templates even more transparent:

```
auto x = std::make_pair( my_matrix.transpose(),
                std::exp( my_matrix, 3 ) );
// decltype(x) = std::pair< Matrix, Matrix >
```

In the absence of such a sophisticated solution, the best thing to do is simply provide a diagnosis if overload resolution selects a function requiring a conversion that did not already happen.

8. Future work

No prototype has been attempted. Implementation is needed for further exploration.

The library needs to be surveyed for getter functions. Most if not all functions named `get`, `begin`, or `end` need lifetime qualification, as do members of iterators.

All the relevant open issues among the various study groups should be rounded up and reviewed.

In particular, at 10 years old, CWG DR 446 and 86 should be reconsidered in modern terms. Objects may conditionally exist within a full-expression and be conditionally destroyed at the semicolon, but per 86, this destruction may not be postponed to the end of the scope. Doing so requires adding persistence to the flags which select among destructors. Given that such flags may persist, we have this:

```
foo && maybe = yeah? move( foo{} ) : move( * (foo*) nullptr );
```

Regardless of the behavior of the indirection operator, should this conditionally put an object on the stack? Should the `move`s be necessary, or are they just irrelevant obfuscation atop already surprising semantics?

9. 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.