Document number: N4147
Date: 2014–09–25
Reply to: David Krauss
(david_work at me dot com)

Inline variables, or encapsulated expressions

1. Abstract

Inline functions have favorable behavior for interfaces which cannot be exposed as objects. Often, users are encouraged to use them to wrap global variables, despite unnatural boilerplate. Other workarounds include class static data members, enumerators, macros, and variable templates, all with awkward syntax or downsides and limited applicability. This proposal defines the `inline` specifier on variable definitions to indicate semantics similar to inline function evaluation and linkage. More generally, this produces a facility for named values, or variables without persistence, which may supersede or complement the various workarounds.

2. Problems solved

These problems render code brittle or encourage nonconforming practices.

2.1. Value-like macro

```
#define NULL 0L
#define SCC_ACK_BIT 5u
#define stdout (& __FDTABLE[ STDOUT_FILENO ])       // OK: rvalue
extern "C" std::FILE * __stdoutp;        // woops, forgot a const
#define stdout __stdoutp      // accidentally modifiable lvalue [1]
#define errno (*__error())     // intentionally modifiable lvalue

if ( std::errno )          // expands to garbage: std::(*error())
    std::fprintf( std::stdout, "Oh noez\n" );  // similar garbage
```

Libraries with C heritage often define globals as macros. Advantages include:

- They are guaranteed not to consume backing storage.
- They are usually not lvalues (unless the expression happens to be), but act as pure values.
- Unlike enumerations, they may be declared piecemeal and with any type.
- Terse syntax.
- Any expression may be evaluated upon use, without writing function-call notation.

[1] From Darwin OS (OS X) `<stdio.h>`

The disadvantages are serious, and most programmers consider macros to be a last resort:

- The macro name leaks into all scopes and namespaces. Ugly nomenclature avoids collisions.
- Names used in the definition must not collide with local scopes.
- They are invisible to semantic analysis, but parse repeatedly and convolute diagnostic messages. Everything suffers from the bloat.
- They are invisible to the optimizer, unless a common subexpression is detected.
- Foreign syntax invites errors in the expression type.
- Easy to forget lvalue/rvalue distinction, e.g. Darwin systems now accept `stdout = 0;`.

With inline variables, the above could be replaced with:

```
inline decltype(auto) NULL = 0L;         // null pointer constant
inline bit_index SCC_ACK_BIT = 5;                        // rvalue
inline auto stdout = & __FDTABLE[ STDOUT_FILENO ]        // rvalue
inline std::FILE * stdout = __stdoutp;                   // rvalue
inline int & errno = *__error();      // reference type => lvalue
```

Not only are all the advantages are retained and all the disadvantages quashed, `inline decltype(auto) NAME = INIT;` provides a drop-in replacement in terms of syntax.

Most of the object-like macros in the standard library's `<cXXXX>` headers could be deprecated in favor of inline variables. This would start toward reducing global namespace pollution, and immediately solve the inconsistency that macro names cannot be namespace-qualified (allowing e.g. `std::errno`). This usage may also provide an important migration path toward modules. Legacy usage during the period of deprecation would be supported with `using` declarations in the global namespace.

2.2. Global constant value

```
struct piecewise_construct_t {};
constexpr piecewise_construct_t piecewise_construct = {};

const int magic_number = 42;

inline std::tuple<int> make_magic() {
    return std::tuple<int>( piecewise_construct, magic_number );
}
```

This function violates the ODR ([basic.def.odr] §3.2/6 [2]) twice because neither of the constructor arguments receives an lvalue-to-rvalue conversion. They are therefore passed by address, but the address depends on the TU because `const` (and `constexpr`) implies internal linkage. Typically this UB is not observed simply because it is uncommon to discriminate object identity.

Header-only libraries have long been gaining popularity, but using a header to define a global variable that is shared among translation units requires some obscure technique. Wrapping a

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

global object as a static local in an inline function is a common solution, but that function cannot be `constexpr` ([dcl.constexpr] §7.1.3/3).

There is no way to share a compile-time constant between TUs unless it is scalar and every use is discarded or undergoes lvalue-to-rvalue conversion. This is a major hole in the language.

Adding `inline` to global constant variables solves the problem within the existing C++ semantic framework. The name of the inline variable evaluates to the value of its initializer, which becomes a temporary object when bound to a reference. The entity ([basic] §3/3) referred to is a value, not an object, which brings the intent and formal semantics in line with the current ODR specification.

For the sake of exposition, here are the proposed fixes:

```
inline constexpr piecewise_construct_t piecewise_construct = {};
inline const int magic_number = 42;
```

The compiler is still free to allocate a reused constant value in statically-initialized storage. Whereas a global constant of literal class type (defined in a header with `const` but not `extern`) currently must generate one copy per TU wherein it is ODR-used, all ODR-uses of an inline variable could be allowed to refer to the same ROMable object. This object would effectively have exactly the same "weak" linkage as an inline function. (The current rule for temporary lifetimes would require inline variable access to construct a copy on the stack. Eliminating this copy might be explored in a follow-up or a separate proposal, as the condition also occurs without inline variables, when users observe addresses of prvalues.)

The details mentioned so far would invalidate usage of such variables as lvalues, e.g. `&std::piecewise_construct`. However, that is exactly what is likely to lead to undefined behavior reflecting non-diagnosis of the ODR or other surprises of internal linkage. Such expressions should be avoided if not deprecated or forbidden outright.

2.3. Global constant table

Observable object identity for constants is needed, though, for global `constexpr` arrays and tables, to allow persistent references to table entries. Currently, it is difficult to convince a `constexpr` value to coexist with cross-TU object identity.

```
// Intuitive:
constexpr literal_array< foo, 300 > make_foomap_table();
constexpr /*static*/ auto foomap_table = make_foomap_table();
// Danger, Will Robinson! Each TU has a different table.

// Correct:
struct foomap_holder {
// Function definition here would be unavailable to initializer.
    constexpr static literal_array< foo, 300 > //No auto allowed.
        foomap_table = make_foomap_table();
};
// In .cpp file:
literal_array< foo, 300 > foomap_holder::foomap_table;
```

The intuitive usage will give the user another separate (but value-identical) table when they add another TU. Pointers into the two tables are unreachable from and incomparable with each other. If an inline function uses the table, it violates the ODR as with the previous example.

The correct usage is difficult to get right. The table must be either a variable template or a `constexpr static` class member. The table size must be hard-coded in a class member declaration because `auto` is not allowed there. (Clang gives this rule a pass.) Any function needed to initialize the member must be defined outside the class, because static member initializers are evaluated while the class is still incomplete, before function definitions are parsed. (Clang gives this a pass too, for template definitions only, potentially at the expense of some determinism.) Finally, one TU must provide a non-initialized definition of the member table, again explicitly stating its type, unless the containing scope is a class template, in which case the definition goes in the header under its own separate `template<…>` qualification.

This proposal makes it easy, with no requirements of class membership or non-membership, no restrictions on `auto`, no out-of-class non-initialized definitions, and no difference between template and non-template usage:

```
inline constexpr auto && foomap_table = make_foomap_table();//OK

struct foomap_holder {
    decltype(auto) make_foomap_table() // OK: function not used
        { return ::make_foomap_table(); } // until table is.

    inline static constexpr auto && foomap_table
        = make_foomap_table(); // OK; this is a definition.
};
```

The reference type indicates that the inline variable produces a glvalue. `constexpr` guarantees that this glvalue is a compile-time constant, which means that it must refer to an object of static storage duration. This object is initialized with the initializer expression, and it is implicitly `constexpr` and `const`. Binding it to an rvalue reference, as shown, produces a `const &&` reference, which behaves the same as a `const &` reference. Pedantic users may prefer to use `auto const &` or no `auto` at all, but it makes little difference.

2.4. Shared global state

The static initialization order fiasco (SIOF) is commonly solved by declaring a global variable as a `static` local in an inline function. However, the global must always be named as a function call. The `inline` specifier isn't directly applicable to object-like variables, but it does improve the usage of an inline function wrapper.

Variable templates and static class data members may potentially provide a future workaround to the SIOF. Currently such objects receive unordered initialization ([basic.start.init] §3.6.2/2; I believe it is a defect that variable templates are not mentioned). They may be defined in multiple TUs, but initialization occurs at a nondeterministic time before entry to `main()`. I have filed a defect report (pending the next list revision) that such objects should be initialized the same way as local static variables, with order relative to the execution of lexically preceding and

succeeding global initializations, but until that is resolved, the problem remains that much more open.

If the user wants shared global state, with private definition (not tied to the interface), which is available before enter to `main()`, then a getter function is needed. An inline reference variable is the ideal way to access such a getter.

```
// global.h
extern foo & get_global();
inline foo & global = get_global();

// global.cpp
foo & get_global() {
    static foo ret;
    return ret;
}
```

Whatever the solution, a library-based global with initialization on demand is going to look like an external function at the ABI level. Allowing evaluation of an identifier to evaluate such a function is a necessary element.

2.5. Member reference to subobject

Occasionally it is useful to keep a reference to part of the immediate object. For example, a CRTP base class will often use the expression `static_cast<derived *>(this)`, which always evaluates to the same thing for the same object. The common solution is to define an accessor function:

```
derived & derived_this()
    { return * static_cast< derived * >(this); }
derived const & derived_this() const
    { return * static_cast< derived const * >(this); }
```

This is a lot of boilerplate and little substance. To be truly complete, `volatile` qualification should be covered as well. The result behaves conceptually like a reference, as the function always returns the same object. However, it requires function call syntax, which can be seen as more boilerplate.

If a member reference were used instead, it would consume storage space, disable the implicit assignment operators, and fail to respect cv-qualification.

A member inline variable allows the repetition to be factored into a generic function:

```
inline auto & derived_this = * preserve_cv_cast<derived>(this);
```

This still requires no storage and does not affect assignability.

2.6. Property accessor

Similarly to the preceding case, we sometimes need accessor functions which do nothing but refer to an easily-accessible resource. The definition of "easy" is certainly controversial, but in practice, accessors usually contain nothing but a member access subexpression.

Many students are taught that accessors are necessary to maintain separation of interface and implementation, and that no class should have `public` data members. There is a grain of truth to this, but defining and using accessors (often with multiple overloads) causes such detriment to readability and maintenance that most experienced C++ programmers do not rigorously follow the rule.

In particular, as a data-like class evolves in complexity from POD to aggregate to refined access protection, converting member accesses to accessors is a jarring interface change. Usually the functionality of an accessor is not sufficiently different from a member access to justify rewriting every use in client code.

Moreover, demanding that an interface comprise only accessors negates the exposition that non-accessor expressions would have, because they are forbidden to exist. A user might be able to tell at a glance that `agg.mem` is O(1) complexity and exception-safe, but that is of no benefit if the syntax seldom occurs in the first place. Conversely, the suspicious character of expressions like `agg.access()` completely blankets a program that exclusively uses accessor-based interfaces, but that is not really a problem. Users rely on a library to behave correctly and perform adequately, but correctness and performance are deep, big-picture issues, and the presence or absence of parentheses does not scratch the surface.

Several popular languages have recently added facilities to perform a function call to access a class member, including C# (2000), Python 2.2 (2001), ECMAScript 5 (2009), and Ruby (1995). Not only do these features exist, but they have been widely accepted as best practice and regarded as an improvement to the prior status quo in their respective communities.

C++ does already support properties, after a fashion, through the more fundamental facilities of user-defined implicit conversion and operator overloading. A nonstatic member "property proxy" may convert to, and assign from, its value type. This solution requires the member to store a reference into its containing object, which breaks the implicit assignment operators and adds unnecessary overhead. Although the average user might understand that such a solution exists in C++, actual implementation is the domain of experts.

Yet, there are also very good reasons for the strong resistance to a C++ properties facility. Our conceptual model of objects and subexpressions is highly evolved, and intricately married to implementation details. Anything like a port of C# syntax would have little uniformity with the rest of the language, entailing a disproportionate increase in semantic complexity. Full-blown properties with specialized getter and setter methods are a non-starter.

This proposal offers a compromise, by merely eliminating the parentheses from getter functions, and the storage overhead and non-assignability from property proxies, and unifying them in a common syntax identical to ordinary member access.

Without inline member variables:

```
struct uuid_aggregate {
```

```
    std::string canonical;

    int version() const
        { return canonical[ 14 ] - '0'; }
};

class uuid_accessor {
    std::array< unsigned char, 16 > data;
public:
    explicit uuid_accessor( std::string in_canonical );

    std::string canonical() const; // getter
    void canonical( std::string const ); // setter
    int version() const;
};

class uuid_proxy {
    typedef std::array< unsigned char, 16 > data_type;
    data_type data;
public:
    class canonical_property {
        data_type & storage;
    public:
        operator std::string () const; // getter
        operator = ( std::string const & ); // setter

        explicit canonical_property( data_type & );
    } canonical { data }; // requires storage as a subobject

    explicit uuid_proxy( std::string const & in_canonical )
        { canonical = in_canonical; }

    uuid_proxy & operator = ( uuid_proxy const & rhs )
        { data = rhs.data; }
};
```

With inline member variables:

```
struct uuid_aggregate {
    std::string canonical;

    inline int version = canonical[ 14 ] - '0';
};

class uuid_accessor {
    std::array< unsigned char, 16 > data;
```

```
    std::string canonicalize() const;
    int extract_version() const;
public:
    explicit uuid_accessor( std::string in_canonical );

    inline std::string canonical = canonicalize(); // getter
    void set_canonical( std::string const ); // setter
    inline int version = extract_version();
};

class uuid_proxy {
    typedef std::array< unsigned char, 16 > data_type;
    data_type data;
public:
    class canonical_property {
        data_type & storage;
    public:
        operator std::string () const;
        operator = ( std::string const & );

        explicit canonical_property( data_type & );
    } inline canonical { data }; // not a subobject

    explicit uuid_proxy( std::string const & in_canonical )
        { canonical = in_canonical; }

    // Implicitly-defined assignment operators work.
}
```

An inline member variable may replace a simple, nullary, inline member function like `uuid_aggregate::version()`, but otherwise it can only wrap self-contained implementation details.

Setter methods like `uuid_accessor::set_canonical` are not much improved by this proposal. Inline variables are most suitable for properties that are immutable or backed by an object, although perhaps not a direct subobject. This covers most of the safe use cases, and avoids the most objectionable, potentially surprising behavior. (In this slightly contrived example, conversion from `std::string` to the UUID class, and subsequent `uuid_accessor` assignment, would likely be preferable to any setter-oriented solution.)

Still, inline variables provide experts with the power of proxy objects, at a lower runtime cost. Also, as temporary objects, a wider variety of design patterns is available, compared to member subobjects which are permanently resident. A proxy derived from the value type would support both value type member accesses, and customized assignment.

To summarize the compromise, inline variables merely enable the tangible benefits of abstract properties, without making them the path of least resistance.

3. Semantics

C++ identifies objects by *names* ([basic] §3/4) which form *id-expressions* ([expr.prim.general] §5.1.1/8). Usually a name maps to a declaration and refers directly to the declared object, but there are exceptions: the name of a reference evaluates to its referent, bound at initialization ([expr] §5/5), and the name of a nonstatic member is transformed into a member access expression ([class.mfct.non-static] §9.3.1/3). These cases may occur in combination: the name of a member reference is first transformed to an access expression and then treated as its referent.

This proposal adds a similar rule: a use of an inline variable evaluates to its initializer, converted if necessary to the declared type. This is similar to function invocation, but simpler in that no sequencing ([intro.execution] §1.9/15) separates the initializer from the context of its use. The implementation is free to actually generate a subroutine, but this is never strictly necessary.

To be clear, an inline variable does not behave like an object. It has no dedicated object representation and no persistence. It is more like a macro which expands to a function call, but safer and more flexible. (A temporary bound to a `constexpr` inline variable declared with reference type does have a persistent object representation, but it is not itself the inline variable.)

3.1. Initialization and evaluation

The initializer of an inline variable is not evaluated at the point of definition, but names used in it are looked up in that scope. It is syntactically an *initializer*, but [dcl.init] §8.5 does not apply when the declaration statement is executed. An inline variable is not itself a named object nor a reference. This proposal will use the terminology *inline variable reference* to refer to an expression naming an inline variable declared with reference type, producing an lvalue or xvalue, and *inline variable object* if declared with non-reference type, which produces a prvalue. The inline variable is not itself a reference or object; the result of evaluation is.

Normally a temporary may be generated for the sake of initializing a reference, but this is inappropriate for inline variable evaluation. An inline variable reference must bind directly ([dcl.init.ref] §8.5.3/5) to its initializer. If the initializer is a prvalue, the inline variable reference evaluation still produces a glvalue. “Perfectly” encapsulating an expression which may have any value category may be accomplished by declaring an inline variable with `decltype(auto)`.

Likewise, copy-initialization semantics ensure that the the implementation may store the initializer and the initialized variable as separate objects, but this is unnecessary and inappropriate for inline variable objects. Copy-initialization of an inline variable object from a prvalue of the same type yields the initializer unadulterated, and the type need not be movable or copyable. If the initializer is a prvalue of a derived type, the result is the base subobject as a prvalue. Otherwise, copy-initialization proceeds with conversions (perhaps only lvalue-to-rvalue conversion), as normal.

Direct-initialization of an inline variable object is performed without any special cases, i.e. by a constructor chosen by overload resolution.

From the user’s perspective, copy-initialization should be used most of the time, to optimally obtain value semantics, and direct-initialization should be used to explicitly access a particular constructor. Applied to references, the two forms of initialization behave identically.

Recursion must be prohibited to avoid meaningless cases. This is simple to check, because inline variable evaluations are expanded in the context of use, even if implemented as subroutines. The implementation limit on "Nesting levels of parenthesized expressions within a full-expression" ([implimits] §B) might be construed to apply, but it deserves explicit specification. Conditional evaluation cannot validate a recursive subexpression in an inline variable initializer.

3.2. `constexpr`

A `constexpr` inline variable is notionally reevaluated per use, but all evaluations are required to have the same result. To ensure applicability of the ODR, its initializer is also evaluated at the point of definition. (See §3.4 *Declaration and definition*.)

A temporary object bound to a `constexpr` inline variable reference is implicitly `constexpr` (and `const`). It must have static storage duration, so the declaration must be `static` or in namespace scope. These considerations perhaps should apply in the non-inline case as well; a core language defect report has been filed. For such a temporary created by a function returning cv-unqualified type, the implementation may add cv-qualification by statically initializing the `const`-qualified object at runtime (program load time) with the object representation determined by the `constexpr` function at compile time.

3.3. Nonstatic members

An inline variable name used as the *id-expression* in a class member access subexpression ([expr.ref] §5.2.5) is substituted directly by its initializer, if the entire initializer is likewise a member name or member function call. (Note that this may be defeated by adding parentheses or braces, which always occur in direct-initialization.) The original object expression ([expr.ref] §5.2.5/3) forms a new member access subexpression with the initializer expression. Otherwise, the entire member access subexpression evaluates to the initializer.

The initializer of a nonstatic member inline variable may use `this`; it evaluates to the address of the object expression and reflects its cv-qualification. An ABI should permit implementations to generate subroutines representing such members, with one subroutine per qualification and value category of the object expression. When such subroutines end up being identical, they may be merged e.g. by the linker, as their addresses cannot be observed. Their purpose would be to reduce executable size or assist debugging; they do not affect the execution model.

Unlike other data members, an inline variable may be declared with placeholder type. This is not a problem because they do not affect class layout, and the placeholder need not be resolved until use. For a nonstatic member, the placeholder type cannot be resolved until the class is complete because this is the scope of its initializer. The type is resolved separately for each use, depending on the value category and cv-qualification of the object expression of the access. Such variables are subject to similar type deduction conundrums as member functions of deduced type, and the same resolutions to those issues should apply uniformly.

`decltype` cannot observe a member inline variable of deduced type in a context where `this` is undefined. Otherwise, the type resolved is that of the implicit member access subexpression. In any case, the special introspective power of `decltype` on declarations never applies to inline variables: it is always identical to the declared type with no adjustment for value category.

A type local to an inline variable initializer, such as a lambda closure type, is the same type across all uses of the variable, except that the identity of such types is unspecified in nonstatic member inline variable initializers. Type variance may be injected by the qualification and category of the object expressions used for access, and it may be removed by conversion to a coalesced subroutine before linking.

An inline variable shall not be initialized by the name of a nonstatic member function; the type system cannot represent such a result. There are no member pointers to inline variables.

3.4. Declaration and definition

The declaration of an inline variable is a definition if and only if it has an initializer. Forward declaration of an inline variable may seem fairly useless because only usage in unevaluated contexts would be viable without a definition. However, the initializer of another inline variable would be such a context. Also, forward declarations tend to model the behavior of member declarations observed in the context of an incomplete class. Therefore inline variable forward declarations are allowed, much like `constexpr` function forward declarations, however subtle the uses may be.

An inline variable is ODR-used if it is potentially evaluated. An inline variable must be defined in every translation unit in which it is ODR-used, before the ODR-use, and the definitions must match to the same degree as do inline function definitions ([basic.odr] §3.2/6).

The default linkage of an inline variable is that of the enclosing scope. The `const` and `constexpr` specifiers do not affect linkage.

The `inline` keyword is currently a function specifier. With this proposal it would make more sense as a storage class specifier. The specification may be adapted with little fuss, as inline variables are stored and linked like inline functions. Like `thread_local`, `inline` may appear with `static` or `extern`. Such a combination has no effect other than to decide the applicability of the ODR. `inline` is applicable in any scope, including block scope. However, combination with `static` or `extern` in block scope is meaningless and forbidden.

Functions do not have object storage, but do have program representation and linkage in an executable “text section.” The value of a `constexpr` inline variable object may be stored in the same way, although a copy may be required if evaluation is required to produce a temporary.

So that an `inline constexpr` variable may have meaningful linkage is (and its initializer is allowed to have backing storage), the initializer must evaluate to the same value at every use and at the point of definition, no diagnostic required. (See §3.1 *Initialization and evaluation*.) This guarantee is provided by the ODR, applied as for inline function and class definitions. Weak linkage of the object representation is appropriate. Read-write access is necessary if there is a mutable subobject and the variable is declared with reference type.

4. Rationale

The described problems all result from undesired consequences of object identity. The workarounds all involve contriving an object whose identity is defined relative to another entity, and perhaps writing an accessor function to return it, with guarantees of writability and persistence according to the const- and reference-qualification of the return type.

The solution is to provide a shortcut to defining such accessor functions, and make it work as much as possible as a name for a value, not for an object or a persistent, immutable reference. Currently, only enumerators have such value-semantic purity.

Function-call semantics carry overhead to support the composition of complex programs, which this facility does not need, such as sequencing side-effects, and requiring an allowance for the implementation to distinguish the return value object from the result of the returned full-expression. So, the behavior is specified in terms of expression evaluation, not function calls.

An inline variable is a means of obtaining a value that may not be identically a specific object. It must be defined only if the value is ever actually obtained, and if used by an inline function, it must always use the same process. So, linkage is specified like that of an inline function.

Use of the `inline` keyword is justified because the user can treat an inline variable almost exactly like a nullary accessor function. The only exceptions are that inline variables:

- may represent an entire member overload set with all reference and cv-qualifications,
- repeal the requirement that a return type be movable or compatible with copy-initialization,
- refuse to generate a temporary for the sake of implicitly converting another temporary,
- cannot potentially re-evaluate to a different result when declared with `constexpr`,
- do not guarantee side-effect sequencing,
- cannot recurse, and
- can be declared in local contexts (although lambdas do this, wrapping an `inline operator()`).

These differences are so minor that most users won't tell the difference, aside from the first item.

4.1. Teachability

Inline variables should be used to provide aliases to things that already exist, or that can be trivially brought into existence, when a reference or named object would be inappropriate due to its object-like persistence or exposure of object identity and storage. For example, they are preferable to unscoped enumerators when integer type is desired.

The initializer of an inline variable should always have low computational complexity, and no effects on program state aside from whatever happens in the course of retrieving the result, such as initializing a global or updating a cache.

On one hand, inline variables provide a means for sneaky side effects. On the other hand, it's more obvious that a side effect is misplaced in an inline variable initializer than in the equivalent inline function definition.

4.2. Alternative: expression aliases and inline object linkage

Richard Smith has informally proposed to cover the ground of this proposal with two features:

```
using name = expression; // expression alias
inline auto global_state = initial_state; // merged definition
```

The `inline` keyword may not be required in the second case; an alternative is to relax the ODR to always allow duplicate definitions. These features are more primitive than inline variables, and they are individually useful, but they have rough edges and do not work together to solve the motivating problems quite as well.

2.1, 2.2. Value-like macro, global constant value

Weakly-linked objects could not efficiently replace a large set of value-like macros unless they could be guaranteed to undergo lvalue-to-rvalue conversion on every use. Passing such a variable by reference would be an ODR-use, requiring it to take space in the link map, which is undesirable especially for systems with little memory.

Expression aliases cannot mention a type without a cast, much like macros. Incorrect type or value category is a common error for macro interfaces, and it should be addressed. (This proposal allows such specification to be avoided explicitly by `decltype(auto)`.

Currently no form of using-declaration is bound by the ODR (except for the corner case of a typedef-name introduced by `using tdn = struct cn`$_{opt}$ `{};`, which may be a misapplication of a C compatibility feature). Unless expression aliases are given linkage, this rules out their naive implementation as subroutines, which may complicate debuggers with little or reduced support for inline functions.

2.3. Global constant table

The alternative proposal would work well in this case, and would avoid the awkward reference syntax. However, this would still require an extension to the ODR to ensure that the same value is generated by each merged definition.

2.4. Shared global state

The shared variable would need to be given unordered initialization, which is the embodiment of the static initialization order fiasco. It is possible to upgrade unordered initialization to something else, by requiring (erstwhile) unordered initializers to be evaluated together with ordered initializers, but only upon the first encounter, as with static local variables. However, this is an untested methodology, and it cannot be as safe as a function returning a static local, which initializes upon access.

Such an upgrade should be considered as a separate proposal, in any case.

The user could fall back on an expression alias to access a global by any arbitrary means, but such interfaces are best specified with explicit type.

2.5, 2.6 Member reference to subobject, property accessor

It is this author's opinion that users should be reminded of the opportunity to specify a type and qualification along with any accessor.

5. Further work

No prototype implementation yet exists. This is instrumental to further progress. Formal standardese may follow.

“First-class” lazy evaluation, without any functor object, is a perennial suggestion on the std-proposals list. This could be represented by decorating function parameters with `inline`. The evaluation semantics may be similar. This would be most useful for inline functions; passing such a variable through any ABI-defined interface would reduce it to a polymorphic functor. The gains are uncertain, though, aside from avoidance of the `[&]{}` notation.

6. Acknowledgements

Gabriel Dos Reis steered me away from proposing weak linkage for `extern constexpr` objects, as it would guarantee object identity were a value semantic is usually more appropriate.

Klaim-Joël Lamotte suggested that member inline variables should depend on the qualification of the object expression used for access.

Richard Smith provided helpful feedback and encouragement on the std-proposals list, although he might not endorse this proposal. He also provided valuable insight into the idea of using a member function in a static member initializer.