Document number: N4149
Date: 2014–09–25
Reply to: David Krauss
(david_work at me dot com)

Categorically qualified classes

1. Abstract

Some classes only work in certain contexts: scope guards are usually useless as subexpressions, and expression template placeholders malfunction as local variables. An unused function result may signal that the caller intends to follow a different protocol, such as with `std::async`. This proposal extends class definitions to prevent such errors, and adds a mechanism to automatically resolve them by type substitution, such as a value type for an expression template. Additionally, generation of non-movable objects becomes more tractable.

The added functionality includes that of the "`auto` evaluation" proposals [1]. This proposal intends to be more expressive, more broadly applicable, and easier to adopt and use.

2. Semantics

```
template< typename cleanup >
class scope_guard & { … };      // & qualifier: never a temporary

template< enum opcode, typename ... operand >
class expression && {  // && qualifier: never a persistent object
    // A conversion applicable to lvalues fixes persistent objs:
    operator double ();
    // Conversions applicable only to temporaries do not apply:
    operator other_expression () &&;
};

template< typename lhs, typename rhs >
expression< opcode::add, lhs, rhs > plus( lhs &&, rhs && );

auto sum = plus( 3.0, 5 );        // Implicitly convert to double
```

Considering the current language, a subexpression naming a preexisting variable is always an lvalue. A subexpression generating a temporary whose life ends at the semicolon is always a prvalue. Since C++11, most users identify lvalues and rvalues with the `&` and `&&` punctuators.

By adding such qualifiers to a class definition as illustrated, objects of the class become forbidden from direct usage as a temporary or a persistent object, respectively.

---

[1] N3748, N4035 *Implicit Evaluation of "auto" Variables and Arguments*, Gottschling, Falcou, Sutter.

Forbidden usage is ill-formed, but the object in question gets a second chance if it may be contextually implicitly converted [2] to another type. Any conversion function will do, even `operator void`, except one that converts to a similarly-qualified class type [3]. To ensure that the author [4] can nominate a particular conversion for this duty, the conversion function lookup ignores any candidate with a ref-qualifier matching the violated class qualifier. The function which is found determines the conversion to perform, but it will not be called except as required by that conversion.

Usage as a persistent object is prohibited by the rvalue qualifier, `&&`. This includes instantiating a complete object which has any storage duration [5] or is an exception object [6].

Usage as a temporary is prohibited by the lvalue qualifier, `&`. *Temporary* in this sense is defined as the complementary opposite of the preceding paragraph, regardless of the official terminology. This constraint is not applied to an expression which is the initializer [7] of an object; an lvalue qualified class may form temporaries for this purpose. This includes function arguments passed by value, constructor calls with one argument passed by value or by reference, and full-expressions in `return` statements. Constraints on the return value itself are applied to the function call postfix-expression, because that context determines its storage duration.

If the qualifiers are used in tandem, `&& &`, both constraints apply. As a result, an object of such a class can only be used as an initializer in conversion (i.e., to represent the value of an object that does not yet exist), or as a subobject. It cannot be persistent nor represent a discarded-value expression, and one can seldom bind to a reference except when passed to a unary constructor.

A temporary of lvalue qualified type is replaced if it has a unique conversion function as described above. It is converted to the type named by the conversion function as if by a `static_cast` expression. The result is used as the value of any expression that the original object had represented. The original object remains a normal temporary in terms of lifetime.

If an object of rvalue qualified, contextually implicitly convertible type initializes a persistent object without a concrete declared type, the type of the latter is determined as if the initializer had the type named by the conversion function. The initialization then proceeds normally, without adjusting the value category of the initializer. In direct initialization, a converting constructor in the deduced type will be used in preference to the conversion function, and the latter may even be deleted.

An object of rvalue qualified type which would be granted storage duration by a reference bound to its subobject must be diagnosed. It would be impossible to apply a conversion.

---

[2] [conv] §4/5. All standard references are to the C++14 FCD, N3936.

[3] This could implicitly invoke two or more user-defined conversions.

[4] This paper will follow a convention that the *user* is the client of a library written by the *author*.

[5] [basic.stc] §3.7. CWG DR 1634 *Temporary storage duration* may create a new storage duration for temporaries, but currently such a concept does not exist.

[6] [except.throw] §15.1/3. Perhaps an “exception storage duration” would also be appropriate.

[7] [dcl.init] §8.5/2

A conversion function which may nominate a class qualification conversion to *cv* `void` or a reference to reference-compatible type shall be defined as deleted. An implementation may wish to warn in other cases where the conversion would still be well-formed if the conversion function were defined as deleted, but it is not.

2.1. Introspection

As it models "by-value argument passing," [8] `std::decay` should include the constraint of rvalue qualification and the associated conversion. If the qualifier exists but not the conversion function, the member typedef `type` shall not exist.

A new metafunction `std::as_temporary<T>` produces a member typename `type` which is

- `T`, if it is not a class type defined with the lvalue qualifier, else
- the return type of the conversion function used to obtain a temporary, else
- it is not declared.

3. Rationale

The qualifiers express a contract which is easily understood by both the library author and the user. Libraries should be able to add class qualifiers without affecting any valid user code, yet specifically fixing or diagnosing invalid usage. Users can read a class interface and tell that it is unsuitable for scoped-variable or subexpression-value semantics. Enforcing this contract, the implementation should provide diagnoses helpfully pointing to the qualifier (and any accompanying library source comment). The syntax packs high-value exposition into an unobtrusive package. The combined `&& &` qualifier is ugly, but its uses are limited to library internals, so users might as well be discouraged from naming or directly using such a class.

The constraints are expressed as prohibitions rather than prescriptions, because the problem to be solved is that certain easily accessible usages produce defects, not necessarily that only certain narrow usages are ever valid. The rules are strong enough to catch all the accidents, without obstructing library implementation.

Since instantiation of objects is constrained, not variable declaration in general, reference binding and perfect forwarding are unaffected, reducing the need for workarounds to restore normal type deduction. Subobjects are unconstrained by class qualification, so if the author needs to remove qualification, a derived wrapper class will do. These traits contrast with the mechanism of `operator auto` and its successors, which must either forbid or permit reference deduction, and suggests that derived classes inherit the conversion semantic.

Opting into the diagnostics should only require adding qualifiers to affected classes and, for expression templates (ETs), their conversion functions. In cases other than ETs, conversions may add new capabilities at incremental cost, such as a guard adding alternative semantics when it is not retained.

---

[8] [meta.trans.other] §20.10.7.6 table 57 (the specification of `std::decay`)

3.1. Exemptions

Subobjects are unconstrained. Although non-static members have names and persist beyond their initialization, a non-static member or base subobject is no more or less temporary than its complete object, from the user's perspective. The complete object is responsible for its own instantiation requirements. Moreover, `auto` deduction of non-static members is not likely ever to be added to C++ [9], so the motivating case of ETs does not apply to members.

Classes that otherwise cannot form temporaries are allowed to do so to initialize other objects. This is in the expectation that the temporary will be converted with move semantics, and after the conversion the temporary will be moved-from, empty, and inconsequential. It is also necessary to support copy-initialization, which is a popular way to initialize persistent objects.

If the initialization exemption is problematic, and a class must truly never form a temporary, the author may hinder such an object from initializing anything. Copy- or move-construction is forbidden naturally by making the class non-movable. Meaningful conversions may be prevented by hiding the value of the class behind access control. Still, the user can leverage the public interface to convert an exempted temporary into a device of their own creation. As for encapsulation, it is probably just as well for the language to err on the side of openness.

If some aspect of temporariness is problematic besides lack of persistence, such as that using a function result in a subexpression is dangerously poor style, implicit conversion to `void` may ensure that is either retained or discarded.

3.2. Conversions

Conversion of an existing object is chosen over outright elimination and substitution of another object. Such a solution would need to instantiate a completely different type from the initializer, necessitating compatible construction semantics in that type. It would be incompatible with existing ET architectures and possibly couldn't solve the ET problem at all. This is a non-starter.

New initialization semantics are avoided. Conversion functions indicate what conversion to attempt, but it still must be accomplished by copy- or direct-initialization. This avoids complicating an already inscrutable part of the language. Also, conversion functions to class type have the bad characteristic of returning by value. Unless it binds to a reference, the return value must be moved, which requires movability and may incur additional overhead.

Because qualification-induced conversion follows conventional rules, the user can always substitute a deduced type for `auto` or `decltype(auto)`, or `static_cast` to a deduced type. This proposal adds no overhead or subtle performance advantage to implicit notation.

If a class is typically used for persistent objects, then rvalue access is unusual, so an rvalue-qualified conversion will be unobtrusive. Such a function has permission to move the contents of its object into a new object. This interface provides a perfect handle to the problem and the perfect environment for its solution.

A conversion function is also used to determine the type of a persistent object with an rvalue-qualified initializer. Likewise, that function is identified by applicability to the forbidden case.

---

[9] N3897 *Auto-type members*, Voutilainen.

Actually calling an lvalue-qualified conversion function with the expectation that it will move from its object may be inappropriate, though. Also, persistent objects are often non-movable, in which case initialization by conversion function is not allowed. Such a conversion (as in ETs) should be either `const` or deleted. Deleted, "`&`" ref-qualified conversions may be odd, but they should never interfere with anything. Destructive conversion may be performed by a corresponding "`&&`" ref-qualified override or by a converting constructor in the deduced type.

Overload resolution fails with ambiguity if a conversion function and a converting constructor are both candidates. Copy-initialization considers conversion functions, but direct-initialization only does so in the absence of a converting constructor. Thus ambiguity can only occur with copy-initialization. In each identified use case, either the constructor is inappropriate, or only direct-initialization is allowed. To interpolate a general rule, defining the conversion function [10] is idiomatic, except for conversion to non-movable types, which do not support copy-initialization and require a converting constructor.

Scenario (example §)	Qualifier	Copy-initialization	Converting constructor in destination class type	Conversion function in qualified class type
Expression template (4.1)	**&&**	To value type	No: Value type may be unaware of ET type.	Defined: This is how typical ETs work.
Scope guard (4.2)	**&**	To unguarded resource type	No: Destination must be movable anyway.	Defined, no ref-qualifier.
Value prototype (4.4)	**&& &**	No: destination is not movable	Yes: Destination is closely integrated with prototype.	Deleted, no ref-qualifier.

Reserving the meaning of conversion functions of lvalues of ET type is less than perfectly elegant. It could possibly break something deep in an ET implementation, but not in the public interface, which never sees intermediate result lvalues. Such breakage can be fixed by replacing implicit conversions of lvalue expressions with explicit constructs. This is stylistically superior.

The lvalue conversion function lookup rule is sufficiently redeemed by its harmlessness and analogous uniformity with the rest of this proposal. In most cases, ET authors may nominate a conversion by omitting the conversion function ref-qualifier, and other applications will not need another conversion function, obviating the need for a ref-qualifier. Nothing untoward is apparent.

3.3. Alternative: conversion upon `auto` deduction

N4035 proposes to use a "`using auto =`" tag to override deduction in non-reference declarators. This is roughly equivalent to the present proposal that a conversion function should decide the deduction of `auto`. In comparison to the present proposal:

1. Only the "fixed persistent variable" case is addressed for ETs and similar value proxies. There is no provision for merely forbidding persistence.
2. The described semantics only cover declaration-statements. Other deduced contexts such as function parameters remain as future work. Other contexts that might disagree with value proxies such as throw-expressions are outside the scope.

[10] Defining, but not as deleted.

3. Reference declarators disable implicit evaluation, even when lifetime extension applies. This fails compatibility with range-for loops. Many users similarly declare `auto &&` "universal references" idiomatically, leaving unfixed cases.
4. It is strictly an added feature, which requires an on/off switch. Although it is designed to fix ill-formed programs, by design it must allow a user to temporarily opt out, because the conversions are not intrinsically limited to cases where their absence would be ill-formed.
5. The tag looks like a member. Although it modifies the usage of the entire class, it may be buried within its definition.
6. As a seeming member, it would be surprising if the tag were not inherited. If a base class uses the feature, a derived class cannot avoid it. For example, an author might wish to derive a class from an ET intermediary and provide backing storage for its internal references to form a self-contained object.
7. The sense of `auto` membership suggests that `foo::auto` may be a valid qualified-id.

The present proposal satisfies the goals and consistency requirements stated in N4035, and goes further with richer functionality and a deeper level of specification.

3.4. Alternative: qualified constructors or destructors

Several times the std-discussion list has seen suggestions that constructors be allowed to discriminate between persistent and temporary usage using ref-qualifiers. Deleting a ref-qualified constructor would prevent the corresponding usage. When both alternatives are implemented, though, the information is more useful at the end of the object lifetime. Also, lifetime extension makes it impossible to know whether a function return value is persistent or temporary.

Applying the same idea to destructors seems a tempting alternative, but it is very dangerous to complicate destruction semantics, and there is still no provision for conversion to another type.

The overload-selection aspect of constructors may also be favorable, to let a class select temporary behavior when constructed from an rvalue. Idiomatically, such is currently done by preferring a factory function interface over a public constructor; different overloads of a factory function like `std::ref` may return differently qualified classes or specializations. If in the future such functions are superseded by a mechanism for deducing class template parameters from constructor arguments [11], this proposal should help that apply to category semantics as well.

[11] EWG 60, N3602 *Template parameter deduction for constructors*, Spertus and Vandevoorde

4. Examples

These are thrown together for the sake of illustration. They are not additional proposals.

4.1. Expression template

```
template< typename lhs_type, typename rhs_type >
class addition &&
    lhs_type & lhs;
    rhs_type & rhs;

    addition( lhs_type & l, rhs_type & r )
        : lhs( l ), rhs( r ) {}
public:
    operator decltype(
            std::declval< std::decay_t< lhs_type > >()
         + std::declval< std::decay_t< rhs_type > >() ) {
        auto evaluated_lhs = std::forward< lhs_type >( lhs );
        auto evaluated_rhs = std::forward< rhs_type >( rhs );
        return evaluated_lhs + evaluated_rhs;
    }

    friend addition add<>( lhs_type &&, rhs_type && );
};

template< typename lhs, typename rhs >
addition< lhs, rhs > add( lhs && l, rhs && r )
    { return { l, r }; }

// Client code

// Usual evaluation.
auto a = add( 1, add( 2, 3 ) );

// Persistent references carry value semantics.
auto && b = add( a, a );

// Pass by value causes immediate evaluation.
template< typename value >
auto double( value v ) { return v + v; }
```

With a mere sprinkling of `&&` qualifiers, any ET class with only a single member conversion function becomes `auto`-friendly, computing values for local variables and pass-by-value parameters.

Separately ref-qualified conversion functions additionally make it safe for the user to define functions over ETs. Destructive or expensive evaluation may be defined only over rvalues to

ensure that each evaluation occurs only once. The above generic addition evaluator is destructive because it forwards its stored operands. Lvalue ref-qualified, non-destructive evaluation may be defined as deleted, or allowed like this:

```
    operator decltype(
            std::declval< std::decay_t< lhs_type > >()
         + std::declval< std::decay_t< rhs_type > >() ) & {
        auto evaluated_lhs = lhs;
        auto evaluated_rhs = rhs;
        return evaluated_lhs + evaluated_rhs;
    }
```

With such support, the user can attempt to refactor ET expressions out of local scope:

```
// Pass by reference preserves ETs; ref-qualifiers add safety.
template< typename evaluable >
decltype(auto) double( evaluable && v )
    { return v + v; } // Error if evaluation is destructive.

template< typename evaluable >
decltype(auto) one_more( evaluable && v )
    { return std::move( v ) + 1; } // Always OK.
```

4.2. `std::async`

The controversy over `async` [12] arises from its return value representing a resource with real economic value, namely a thread of execution consuming memory and processor cycles. (Pedantically, it represents the result of the thread.) Users are safer if the thread or process calling `async` cannot be hijacked to spawn endless new threads. Furthermore, when `async` is used with lambda captures by reference from a local scope, the lambda object is invalid when that scope exits. There is a motivation to provide "local" threads which aren't *too* asynchronous.

The current solution is to add a flag to the result object, triggering `wait` synchronization (like `join`) upon destruction. Since this flag is part of the object's value, it may be exported to another scope or thread, significantly weakening the safety guarantees, and adding unexpected, nondeterministic locking patterns, leading to deadlock if an `async` task owns its own `future`. (For example, the user may wish to use `future<void>` for selective synchronization.)

The `wait`-on-destroy semantic also embarrassingly and confusingly erases the gains of multithreading when the result is not retained, so that it gets destroyed immediately.

---

[12] N3451 *async and ~future*, Sutter, N3780 *Why Deprecating `async()` is the Worst of all Options*, Josuttis, and various other papers in the 2013-05 and 2013-09 mailings.

A solution enabled by this proposal would be a new lvalue-qualified class, encapsulating the `wait` but only applying to the local scope. This semantic can be moved to another guard, but vanishes when ownership is transferred to a temporary. Furthermore, such a transfer occurs implicitly if no guard existed in the first place.

```
template< typename Result >
struct guarded_future
    & // If named, the guard scope persists as long as the task.
    // If temporary, decay to a normal future.
    : future< Result > {
    operator future< Result > () &&
        { return std::move( * this ); }

    ~ guarded_future()
        { if ( valid() ) wait(); }
};
```

In addition to conventional, local guards, this class is suitable for the heap as well. However, it should be noted that a `future` (temporary or not) should never initialize a `guarded_future`, especially one on the heap, because any exception such as `bad_alloc` would destroy the unguarded object and break the guard guarantee.

4.3. Scope guard

Scope guards are afflicted with the possibility that the user will forget to declare a variable.

```
make_guard( [&]() noexcept { work_items.clear(); } ); // oops
```

The author can request a diagnostic with the likes of `__attribute__((warn_unused_result))`, but unless the programmer already understands the problem, his reflex may be to idiomatically suppress the warning with a cast to `void`.

This is handily fixed with an lvalue qualifier:

```
struct scope_guard_base {};
typedef scope_guard_base && scope_guard;

template< typename cleanup >
auto make_guard( cleanup in ) {
    class scope_guard_impl
        & // Usage: “scope_guard g = make_guard( … );”
        : public scope_guard_base {
        cleanup action;
    public:
        scope_guard_impl( cleanup in )
            : action( std::move( in ) ) {}
        scope_guard_impl( scope_guard_impl && ) = delete;
        ~ scope_guard_impl ()
            { action(); }
```

```
    };
    return { std::move( in ) };
}
```

Any invalid usage is an error, and the diagnostic should print the line from the library containing the `&` qualifier, which should explain the issue and correct usage.

4.4. Graph node

Node objects in directed graph structures are non-movable without a list of incoming edges. Even if this condition is met, extraneous move operations will slow an intensive graph algorithm, and they can seldom be completely optimized away when neighboring nodes are modified.

Graphs are not particularly programmer-friendly, but C++ forces an additional trade-off:

1. Implement non-movable nodes on the heap. The user cannot have a local node variable but only a node handle, usually a naked pointer. Any function can produce a node, but SBRM is impossible and allocator overhead is incurred per node.
2. Implement movable nodes. Each node contains an additional list of incoming edges, which are adjusted to reflect move operations. The allocator overhead is replaced with a likely greater cost in time and space.
3. Implement non-movable nodes with no factory functions. Constructing a node requires supplying all its raw ingredients to a constructor. This process cannot be encapsulated. This is compatible with #1 but the interface is usually only suitable internally to the library.
4. Implement non-movable nodes with two-phase initialization. A factory function may return a node, but it is not validated with incoming edges until the client confirms that the object has been moved into its final location. In the meantime, a description of the incoming edges must be hidden within the object.

#3 and #4 are the most efficient, but unsafe and onerous to use. It would be nice to have an idiom for a class that describes the values of another class, without troubling the user about multi-step initialization. Enter the `&& &` prototype qualifier:

```
typedef std::vector< struct graph_node * > node_list;

struct graph_node_descriptor
    && & // This class generates graph_nodes. Not for direct use.
    {
    node_list incoming, outgoing;

    operator graph_node () = delete;
};

struct graph_node {
    std::size_t source_count;
    node_list outgoing;
```

```
    graph_node( graph_node_descriptor && in )
        : source_count( in.incoming.size() )
        , outgoing( std::move( in.outgoing ) ) {
        for ( auto * source : in.incoming ) {
            source.outgoing.push_back( this );
        }
    }

    graph_node( graph_node && ) = delete;

    ~ graph_node() {
        for ( auto * sink : outgoing ) -- sink.source_count;
        if ( source_count != 0 ) {
            // There's an indestructible node out there.
            try { std::cerr << "dangling sources\n" ); }
            catch (...) {}
            std::abort();
        }
    }
};

// Factory function.
graph_node_descriptor make_root( graph & g ) {
    node_list outgoing;
    for ( graph_node & n : g ) {
        if ( n.source_count == 0 ) outgoing.push_back( & n );
    }
    return { {}, std::move( outgoing ) };
}
```

The factory function `make_root` can initialize a local node object, a subobject, or a parameter passed by reference. (Passing by value would be impossible anyway.) The user never needs to know about the `graph_node_descriptor` class, except perhaps to define a custom factory function.

5. Further work

This proposal has not yet been prototyped. The next step is to implement it and test with popular ET and scope guard libraries. Formal standardese may follow.

`operator void` is required here to be deleted, but it may possibly represent a useful feature. The effect of defining (and calling) `operator void` would be similar to that of converting to a proxy class which takes some action in its destructor. The difference is whether that action occurs upon an exception, during unwinding. Prototypes might consider allowing nontrivial `operator void`, to see if something may be done with it.

When a given class definition may require qualification depending on template parameters, conditional qualification could be helpful. This proposal suggests deriving a qualified class and adapting the interface as necessary — if that means no adjustment, the derived class only declares forwarding constructors. It may be cleaner to offer a Boolean hook, as there is for function `noexcept(`*constant-expression*`)` specifiers.

Class qualification with `const` might also be interesting, particularly for a class with only implicit constructors and destructors. Such a class would never be observed to be non-`const` by the user, so the implementation could safely assume that every instance is immutable. This may be investigated in a followup proposal.

6. Acknowledgements

Ville Voutilainen helpfully guided me away from the dead end of destructor overloading.

Falcou, Gottschling, and Sutter's work on `operator auto` inspired the extension of this idea beyond scope guards.