US22/DE9 Revisited: Decltype and Call Expressions

ISO/IEC JTC1 SC22 WG21 N3276=11-0046 - 2011-03-23

Eric Niebler (eric.niebler@gmail.com)
Doug Gregor (doug.gregor@gmail.com)
James Widman (widman@gimpel.com)

This paper deals with US22/DE9, which mistakenly attributed a type-completeness requirement to any expression argument of `decltype`, but which was nevertheless trying to call attention to a real problem: that when the expression of a *decltype-specifier* is a function call expression, the longstanding requirement for a call-expression's type to be complete (from 5.2.2 [expr.call]) is both unnecessary and harmful. In that limited context, this old requirement causes unexpected and catastrophic problems. The committee decided in Rapperswil that this issue was NAD. The authors of this paper believe the issues caused by that requirement are real and very serious, possibly leading to poor adoption of `decltype` and `result_of`.

This paper will do the following:

- Demonstrate the seriousness of the issue.
- Give real-world experience of the problem as encountered in Boost.
- Suggest core wording changes that will address the problem.
- Weigh the pros and cons of accepting this resolution vs. others.
- Present the experience of a compiler writer who implemented the suggested resolution in clang.

Problem

The most common intended use of `decltype` is as the return type of a function template, but it behaves differently than just writing the type directly. From rationales given in the papers that introduced `decltype` (N1978) and `result_of` (N1454, N2194), it is clear that this equivalence was expected to hold. [1] That fact that it does not causes code to break.

Trivial Example

Here is a simple program that demonstrates the problem:

```
template<class T> struct S;

template<class X, class Y> struct pair {};

template<class T> S<T> wrap(T) { return 0; }

template<class T>
struct S
{
```

```
    S(int = 0) {}

    decltype(wrap(pair<T,T>())) foo() { return 0; } // ERROR

    S<pair<T,T> > bar() { return 0; }               // OK
};

S<int> s;
```

The member functions `S::foo` and `S::bar` should be equivalent. However, the decades-old rule (from 5.2.2 [expr.call]) that requires the completeness of the type of the expression `wrap(pair<T,T>())`, used in the `decltype()` in the declaration of `S::foo()`, still applies (even though the old motivations for type completeness---namely the need to allocate storage for the prvalue temporary and the need to consider invoking the type's destructor for that temporary---do not exist in this new context). This sets off a cascade of template instantiations that runs the compiler out of memory (tested on MSVC 10 and Comeau Online as of 1/6/2011). The declaration of `S::bar` does not. The implications for new code using `decltype` and, by extension, old code using TR1's `result_of` are dire and cannot be worked around.

Background

TR1 introduced the `result_of` facility as a way for generic code to compute the return type of any callable. It is designed to work even with callables whose return types depend on their argument types in non-trivial ways. To handle this case in the absence of `decltype`, TR1's `result_of` resorted to an idiom: a nested result template that `result_of` could use to perform the computation. This meant extra work for authors of function objects, and could be considered a necessary evil in C++03. TR1's `result_of` was explicitly designed as a bridge to the coming `decltype` version, which would eliminate the need for this extra work.

`decltype` was introduced in large part to ease the declarations of functions whose return types depend on their argument types in non-trivial ways. In fact, that was the only rationale given for `decltype` in N1978 (Section 2.1). Regrettably, this is precisely the situation in which the problem is most likely to occur.

N2194 changed `result_of` to use `decltype`, obviating the need for nested result templates. `decltype` is a step forward for `result_of` in terms of usability. Early industry experience, however, has shown that this transition is problematic.

The Problem in Detail

The problem has to do with templates, instances of which can be mentioned in code without causing their instantiation. For instance:

```
template<class T> foo { };
typedef foo<int> foo_int; // foo<int> only mentioned here
foo_int f;                // foo<int> instantiated here
                          // because the size & alignment requirements
                          // of foo<int> are needed because storage for
                          // a foo<int> must be allocated; and because
                          // a  default constructor of foo<int> might
                          // be non-trivial (and if so needs
                          // to be called implicitly);
                          // and because the destructor of foo<int> might
                          // be non-trivial (and if so needs to be called
                          // implicitly at the end of the lifetime of f).
```

As Relates to TR1 `result_of`

The old TR1 `result_of` implementation works with nested result templates that expose the return type as a nested typedef:

```
struct make_foo
{
    template<class Sig> struct result;
```

```
        template<typename This, typename T>
        struct result<This(T)>
        {
            typedef foo<T> type; // foo<T> mentioned here
        };

        template<class T>
        foo<T> operator()(T t) // foo<T> instantiated at call site
        {
            return foo<T>();
        }
    };
```

Notice in the above code that within the nested result template, `foo<T>` is only mentioned, and hence is not instantiated. Only when `foo::operator()` is actually called is `foo<T>` required to be complete. It follows that in TR1, the following does not cause any instantiation of `foo`:

```
// In TR1, foo<int> is not instantiated:
typedef std::tr1::result_of<make_foo(int)>::type foo_int;
```

As Relates to C++0x `result_of`

The C++0x version of `result_of` uses `decltype` to compute return types, eliminating the need for nested result templates. The following line of C++0x code:

```
typedef std::result_of<make_foo(int)>::type foo_int;
```

is roughly equivalent to[2]:

```
typedef decltype(make_foo()(int())) foo_int;
```

The problem is that function call expressions require their return types to be complete. As a consequence, the above use of `result_of` now causes an instantiation of `foo<int>`, whereas in TR1 it did not.

Timing is Everything

Why should we be concerned about instantiating a template eagerly when presumably we're going to instantiate the template anyway at some other point? By causing an eager template instantiation, we create the potential for an infinite series of cascading template instantiations that causes compilation to fail where formerly it succeeded. This happens in code where `result_of` is used to compute the return type of a member function of a class template and the resulting type is itself an instance of that class template, as happened with the simple example given above.

A More Realistic Example

The code above is awkward and unlikely. Appendix A presents a more realistic example. It is the skeleton of a very simple library for lazy expression evaluation using expression templates.

Expression templates are fairly mainstream technique used in high-performance libraries like Blitz++ and in domain-specific libraries like Boost.Lambda and Boost.Spirit. It makes use of overloaded operators that, rather than eagerly evaluate their results, instead build a tree representing the expression so that it can be evaluated later.

Appendix A presents a very simple expression template library implemented in a straight-forward manner. Nodes in the expression tree are represented by instances of the `Expr` template. New nodes are created by operator overloads defined as members of `Expr`. The return types of the operator overloads are computed using `result_of` and the `MakeExpr` function object. With the TR1-version of `result_of`, everything works. With the `decltype` version it doesn't; that is, when `BUGBUG` is defined, the code

in Appendix A causes Visual C++ 10 and Comeau's online compiler to recursively instantiate an infinite number of templates.

The Implications for Generic Code

The implications of US22/DE9 are plain: `decltype`, and hence `std::result_of`, cannot safely be used in class templates to compute the return types of member functions. Note that the primary mandate of both `decltype` and `std::result_of` is to make it easy to compute return types in generic code. Due to the context-insensitivity of the old type-completeness rule in 5.2.2 [expr.call], `decltype` and `std::result_of` both fail their primary mandate.

The other implication is that some valid C++03 designs that are currently making use of `std::tr1::result_of` must eschew `std::result_of` for fear of being bitten by this problem. This will lead to a bifurcation in the library space: some code will move to `std::result_of`, and some will stay with `std::tr1::result_of`. Those libraries that stick with `tr1::result_of` impose the TR1 `result_of` protocol on all its users. And as TR1 is not a standard, there is no guarantee that `std::tr1::result_of` will continue to be available.

Note that at least one major vendor currently shipping both `std::result_of` and `std::tr1::result_of` simply share the same `decltype`-based implementation[3], so this problem can bite even those users that choose to stick with `std::tr1::result_of`.

Field Experience: Boost

This is not some dreamed up and unlikely scenario. The problem was discovered in the field by porting `boost::result_of` to use `decltype` on those compilers that support it. When this change was made, the tests of one important foundational library, Boost.Proto, and all the libraries that depend on it (Boost.Spirit, Boost.Xpressive) began failing.

To work around the problem, Boost has had to ship two versions of `result_of`: one that uses the TR1 protocol (`boost::tr1_result_of`), and one that uses `decltype` if it's available (`boost::result_of`). Boost.Proto had to be changed to use `boost::tr1_result_of` in all places where it was determined this problem could crop up. And at the time of writing (1/6/2011), the use of `decltype` in `boost::result_of` is turned off by default until this issue is resolved.

This has downstream effects: all Boost users are still bound to use the TR1 protocol for their function objects even when `decltype` is available. We are not seeing the much hoped-for reduction of meta-programming promised by `decltype` and `std::result_of`. Said Doug Gregor, author of the TR1 `result_of` proposals, in a private exchange:

> It's a personal embarrassment, because `result_of` was meant to be a bridge to `decltype`, and it's completely broken by this. […] It's definitely a problem that return-type-computing metaprograms will be forced to live on, when we tried so hard to kill them with `decltype`. Will `decltype` even get used?

Solution

Notes About the Proposed Wording Change

This change makes it so that, when the expression in a *decltype-specifier* is a function call expression, and when the return type of the selected function is a class type,

- the type is not required to be complete and
- a temporary object is not introduced for the return value (and as a result, storage is not allocated for the prvalue and a destructor is not invoked).

Both changes occur in section 5.2.2 [expr.call] and reference 12.2 [class.temporary]. 12.2 p1 is modified to clarify that, wherever a temporary is introduced in an unevaluated context, semantic constraints are checked.

Note that sub-expressions (e.g. function calls used as call arguments) are not affected. Also, unevaluated operands outside of a `decltype` are not affected. For example, the following code was ill-formed in C++03 and will remain ill-formed: *[Example:*

```
struct A g();
char f(struct B);
int f(...);

unsigned long z = sizeof(f(g())); // Error: return type of g() must be
                                  // complete.
```

— end example]

This behavior must be maintained because the outcome of overload resolution depends on conversions and therefore type-completion.

If the user wants type completion, they can wrap the expression in a call to an "identity" function template. (See `i()` in the example in the proposed wording below. This resolves the issue raised in c++std-core-16953.)

Proposed Resolution

- Change 3.10 basic.lval p4 as indicated:

> 4. Class prvalues can have cv-qualified types; non-class prvalues always have cv-unqualified types. **Unless otherwise indicated (5.2.2),** ~~P~~**p**rvalues shall always have complete types or the void type; in addition to these types, glvalues can also have incomplete types.

- Change 5.2.2 expr.call p3 as indicated:

> 3. If the postfix-expression designates a destructor (12.4), the type of the function call expression is void; otherwise, the type of the function call expression is the return type of the statically chosen function (i.e., ignoring the virtual keyword), even if the type of the function actually called is different. This type shall be a**n** ~~complete~~ object type, a reference type or the type void.

- Append a new paragraph to the end of 5.2.2 expr.call:

> 10. A function call is an lvalue if the result type is an lvalue reference type or an rvalue reference to function type, an xvalue if the result type is an rvalue reference to object type, and a prvalue otherwise.
> 11. **If a function call is a prvalue of object type:**
>
> - **If the function call is either**
>
> - **the operand of a *decltype-specifier* or**
> - **the right operand of a comma operator that is the operand of a *decltype-specifier*,**
>
> **a temporary object is not introduced for the prvalue. The type of the prvalue may be incomplete. *[Note:* As a result, storage is not allocated for the prvalue and it is not destroyed; thus, a class type is not instantiated as a result of being the type of a function call in this context. This is true regardless of whether the expression uses function call notation or operator notation (13.3.1.2 over.match.oper). — *end note] [Note:* Unlike the rule for a *decltype-specifier* that considers whether an *id-expression* is parenthesized (7.1.6.2), parentheses have no special meaning in this context. — *end note]***
>
> - **Otherwise, the type of the prvalue shall be complete.**

- Append a new paragraph to the end of 7.1.6.2 dcl.type.simple:

> 5. ***[Note:*** **In the case where the operand of a *decltype-specifier* is a function call and the return type of the function is a class type, a special rule (5.2.2) ensures that the return type is not required to be complete (as it would be if the call appeared in a sub-expression or outside of a *decltype-specifier*). In this context, the common purpose of writing the expression is merely to refer to its type. In that sense, a *decltype-specifier* is analogous to a use of a *typedef-name*, so the usual reasons for requiring a complete type do not apply. In particular, it is not necessary to allocate storage for a temporary object or to enforce the semantic constraints associated with invoking the type's destructor.** ***[Example:***
>
> ```
> template<class T> struct A { ~A() = delete; };
>
> template<class T> auto h()
>     -> A<T>;
>
> template<class T> auto i(T)  // identity
>     -> T;
>
> template<class T> auto f(T)  // #1
>
>     -> decltype(i(h<T>()));  // forces completion of A<T> and implicitly uses A<T>::~A() for the
>                              // temporary introduced by the use of h(). (A temporary is not
>                              // introduced as a result of the use of i()).
> ```

```
template<class T> auto f(T) // #2
    -> void;

auto g() -> void {
    f(42);  // Ok; calls #2. (#1 is not a viable candidate: type deduction fails (14.8.2) because
            // A<int>::~A() is implicitly used in its decltype-specifier)
}


template<class T> auto q( T )
  -> decltype((h<T>()));   // Does not force completion of A<T>;
                           // A<T>::~A() is not implicitly used within the
                           // context of this decltype-specifier.

void r() {
    q(42);  // Error: deduction against q succeeds, so overload resolution
            // selects the specialization "q(T)->decltype(h<T>()) [with T=int]".
            // The return type is A<int>, so a temporary is
            // introduced and its destructor is used, so the program
            // is ill-formed.

}
```

— end example] — end note]

- Change 12.2 class.temporary p1 as indicated:

> 1. Temporaries of class type are created in various contexts: binding a reference to a prvalue (8.5.3), returning a prvalue (6.6.3), a conversion that creates a prvalue (4.1, 5.2.9, 5.2.11, 5.4), throwing an exception (15.1), entering a handler (15.3), and in some initializations (8.5). *[Note:* the lifetime of exception objects is described in 15.1. *— end note]* Even when the creation of the temporary object is **unevaluated (Clause 5) or otherwise** avoided (12.8), all the semantic restrictions shall be respected as if the temporary object had been created **and later destroyed**. *[Note:* Even if ~~the copy/move constructor is not called~~ **there is no call to the destructor or copy/move constructor**, all the semantic restrictions, such as accessibility (Clause 11) **and whether the function is deleted (8.4.3)**, shall be satisfied. **However, in the special case of a function call used as the operand of a *decltype-specifier* (5.2.2), no temporary is introduced, so the foregoing does not apply to the prvalue of any such function call.** *— end note]*

[Drafting note: 12.2 class.temporary p3 is unchanged. In drafts of this proposal, it was preceded by an exception for the `decltype` case. But p3 applies only ``When an implementation introduces a temporary object'', and the change proposed for 5.2.2 above ensures that a temporary is *not* introduced for the *expression* of a *decltype-specifier*, so the destructor call is naturally suppressed.]

Options Considered and Dismissed

Jason Merrill pointed out that the wording change proposed above introduces a very special case, so we considered a more general rule: in this alternative, the type of a function return value is required to be complete only if

- the call is potentially evaluated or
- the return value (a prvalue) is converted.

(An identity-converted value can be regarded as "converted", since some types are not copyable/movable.)
For reasons mentioned above (see the `sizeof` example), it's important to allow type-completion on any function return value that is converted. So to see where this alternative rule would prevent type-completion we should consider operands that are *not* converted. We find them by elimination:

- We can rule out every operand that is required to be of scalar type because that would imply a user-defined conversion to the scalar type.
- The Lvalue-to-rvalue conversion is always applied to operands 2 and 3 of the conditional operator even if they are of identical class type and have the same value category.
- The address operator does not apply because it cannot have a prvalue operand.
- `sizeof` and `alignof` always need a complete type.
- Cast operators seem pointless to even contemplate for `decltype` purposes.
- `typeid` requires a complete type to see whether its operand is of polymorphic type.

So we are left with:

- a full-expression;
- the left operand to comma;
- the right operand to comma;
- the *expression* in a *noexcept-expression*.

As it turns out, it can be useful to write `decltype(e1,e2)` where type-completion is desired for e1, so we do not want to suppress instantiations there. In the case of a *noexcept-expression*, the implicit invocation of the return type's destructor on the temporary is probably desired because it might throw, in which case the *noexcept-expression* is would be `false`. By contrast, an implicit destructor call would not affect the type of a *decltype-specifier*.

Implementation Experience

We have implemented our proposed resolution for the Clang compiler. The implementation itself is simple and straightforward, suppressing the type-completion and temporary-construction logic when building calls at the top level of a `decltype` expression. We tested the implementation against Boost's Proto library (configured to use `decltype` via `result_of`), confirming that the proposed resolution does in fact address the issues seen in Boost and does not appear to cause any breakage. We do not anticipate that other implementors will have any difficulty implementing this change.

Acknowledgements

Many thanks to Jason Merrill, Daniel Krügler, Steve Adamczyk, and Jens Maurer for early reviews and helpful comments.

Appendix A: Source Code for a Realistic Example

The following code demonstrates how reasonable and valid code using TR1 `result_of` can stop working when `result_of` is switched to use `decltype`. It is the skeleton of a simple expression template library. An expression is captured in a tree of `Expr` objects, built by operator overloads defined as member functions of the `Expr` template. It makes idiomatic use of a `MakeExpr` function object for computing return types. Porting this code to C++0x's `result_of` will break this perfectly valid and reasonable design.

```
// Uncomment this for a wild time:
//#define BUGBUG

// A simplified result_of implementation.
// If BUGBUG is defined, it uses decltype.
// Otherwise, it uses the TR1 result_of
// protocol.
template<typename Sig>
struct result_of;

#ifdef BUGBUG
  template<typename T> T& declvar();

  // use decltype
  template<typename Fun, typename T>
  struct result_of<Fun(T)>
  {
    typedef decltype(declvar<Fun>()(declvar<T>())) type;
  };

  template<typename Fun, typename T, typename U>
  struct result_of<Fun(T, U)>
  {
    typedef decltype(declvar<Fun>()(declvar<T>(), declvar<U>())) type;
  };
#else
  // use TR1 protocol
```

```
  template<typename Fun, typename T>
  struct result_of<Fun(T)>
    : Fun::template result<Fun(T)>
  {};

  template<typename Fun, typename T, typename U>
  struct result_of<Fun(T, U)>
    : Fun::template result<Fun(T, U)>
  {};
#endif

// simple tuple type
template<typename A0 = void, typename A1 = void, typename A2 = void>
struct tuple;

template<typename A0>
struct tuple<A0, void, void>
{
  A0 a0_;
  tuple(A0 const &a0) : a0_(a0) {}
};

template<typename A0, typename A1>
struct tuple<A0, A1>
{
  A0 a0_;
  A1 a1_;
  tuple(A0 const &a0, A1 const & a1) : a0_(a0), a1_(a1) {}
};

// A node in an expression tree
template<class Tag, class Args> // Args is a tuple.
struct Expr;

// A function object that builds expression nodes
template<class Tag>
struct MakeExpr
{
  template<class Sig>
  struct result;

  template<class This, class T>
  struct result<This(T)>
  {
    typedef Expr<Tag, tuple<T> > type;
  };

  template<class This, class T, class U>
  struct result<This(T, U)>
  {
    typedef Expr<Tag, tuple<T, U> > type;
  };

  template<class T>
  Expr<Tag, tuple<T> > operator()(T const & t) const
  {
    return Expr<Tag, tuple<T> >(tuple<T>(t));
  }

  template<class T, typename U>
  Expr<Tag, tuple<T, U> > operator()(T const & t, U const & u) const
```

```
  {
    return Expr<Tag, tuple<T, U> >(tuple<T, U>(t, u));
  }
};

// Here are tag types that encode in an expression node
// what operation created the node.
struct Terminal;
struct BinaryPlus;
struct FunctionCall;

typedef MakeExpr<Terminal>      MakeTerminal;
typedef MakeExpr<BinaryPlus>    MakeBinaryPlus;
typedef MakeExpr<FunctionCall>  MakeFunctionCall;

template<class Tag, class Args>
struct Expr
{
  Args args_;
  explicit Expr(Args const & t) : args_(t) {}

  // An overloaded operator+ that creates a binary plus node
  template<typename RTag, typename RArgs>
  typename result_of<MakeBinaryPlus(Expr, Expr<RTag, RArgs>)>::type
  operator+(Expr<RTag, RArgs> const &right) const
  {
    return MakeBinaryPlus()(*this, right);
  }

  // An overloaded function call operator that creates a unary
  // function call node
  typename result_of<MakeFunctionCall(Expr)>::type
  operator()() const
  {
    return MakeFunctionCall()(*this);
  }
};

int main()
{
  // This is a terminal in an expression tree
  auto i = MakeTerminal()(42);

  i + i; // OK, this creates a binary plus node.

  i(); // OK, this creates a unary function-call node
}
```

---

[1] Below are quotes from the various proposals showing that `decltype` and `result_of` were intended to solve problems with function return type declarations, and that the intention was that the `decltype` formulation of return types was intended to be functionally equivalent to just writing the type directly.

> *From N1454*:
> "result_of is intended to act as a bridge from the function objects of C++98 to more powerful function objects as used in current binding and composition libraries. This bridge provides both forward compatibility, allowing C++02 code to derive benefits from future C++ revisions without modification"
>
> *From N1978*:
> "2.1 Why decltype is crucial: The return type of a generic function often depends on the types of the arguments. In some cases the dependency can be expressed within the current language. [...] In other cases this is not as easy, or even possible."
>
> *From N2194*:

> "In particular the result_of hook---which is the only aspect of the Standard Library that this document changes—was designed with forward-compatibility in mind [1]. result_of currently says that implementations are permitted to get the return type of a particular function call by any means possible, so long as they get the answer right; if they cannot do so, result_of specifies a protocol that the implementation should follow to extract the return type from library- and user-provided information. With decltype, every implementation can get the answer right, so we need only eliminate the weasel-wording result_of currently uses. We note that a C++0x result_of meets the requirements of a TR1 result_of."

[2] This ignores the use of tricks to avoid the default-constructability requirements imposed here, but that's not relevant to this discussion.

[3] Microsoft Visual Studio C++ 10.0