Decltype (revision 6): proposed wording

Programming Language C++
Document no: N2115=06-0185

Jaakko Järvi
Texas A&M University
College Station, TX
jarvi@cs.tamu.edu

Bjarne Stroustrup
AT&T Research
and Texas A&M University
bs@research.att.com

Gabriel Dos Reis
Texas A&M University
College Station, TX
gdr@cs.tamu.edu

2006-11-05

1 Introduction

We suggest extending C++ with the `decltype` operator for querying the type of an expression. This document is a revision of the documents N1978=06-0048 [JSR06], N1705=04-0145 [JSR04], 1607=04-0047 [JS04], N1527=03-0110 [JS03], and N1478=03-0061 [JSGS03], and builds also on [Str02]. The document reflects the specification as discussed in the EWG in the Portland meeting, October 2006. Changes from the previous revision [JSR06] include the following:

- The specification now takes rvalue references into account
- Parenthesized *id-expression* inside `decltype` is not considered to be an *id-expression*.
- Editorial changes to the wording.
- The new function declaration syntax

  ```
  auto f(params) -> return-type
  ```

  that moves the return type after the function's parameter list is not part of the wording. There will be a separate document for that functionality.
- Most of the motivation, background, and history discussion has been dropped—we only include wording, and examples of the implications of the proposed rules.

2 The `decltype` operator

2.1 Syntax of `decltype`

The syntax of `decltype` is:

```
simple-type-specifier
  ...
  decltype ( expression )
  ...
```

We require parentheses (as opposed to `sizeof`'s more liberal rule). Syntactically, `decltype(e)` is treated as if it were a *typedef-name* (cf. 7.1.3). The operand of `decltype` is not evaluated.

2.2 Semantics of `decltype`

Determining the type `decltype(e)` build on a single guiding principle: look for the declared type of the expression `e`. If `e` is a variable or formal parameter, or a function/operator invocation, the programmer can trace down the variable's, parameter's, or function's declaration, and find the type declared for the particular entity directly from the program text. This type is the result of `decltype`. For expressions that do not have a declaration in the program text, such as literals and calls to built-in operators, lvalueness implies a reference type.

The semantics of the `decltype` are captured with the following rules (these rules are directly from the proposed wording given in Section 3). Note that *rvalue references* (see N1952=06-0022 [Hin06]) are taken into consideration in these rules. Any case producing an rvalue reference type falls under the rule 2 below.

> The type denoted by `decltype(e)` is defined as follows:
>
> 1. If `e` is an *id-expression* or a class member access (5.2.5 [expr.ref]), `decltype(e)` is defined as the type of the entity named by `e`. If there is no such entity, or `e` names a set of overloaded functions, the program is ill-formed.
> 2. If `e` is a function call (5.2.2 [expr.call]) or an invocation of an overloaded operator (parentheses around `e` are ignored), `decltype(e)` is defined as the return type of that function.
> 3. Otherwise, where `T` is the type of `e`, if `e` is an lvalue, `decltype(e)` is defined as `T&`, otherwise `decltype(e)` is defined as `T`.
>
> The operand of the `decltype` operator is not evaluated.

2.3 Decltype examples and discussion

In the following we give examples of `decltype` with different kinds of expressions. First, however, note that unlike the `sizeof` operator, `decltype` does not allow a type as its argument:

```
sizeof(int);     // ok
decltype(int);   // error (and redundant: decltype(int) would be int)
```

2.3.1 Variable and function names

- Variables in namespace or local scope (rule 1 applies):

```
int a;
int& b = a;
const int& c = a;
const int d = 5;
const A e;

decltype(a)    // int
decltype(b)    // int&
decltype(c)    // const int&
decltype(d)    // const int
decltype(e)    // const A
```

Note that parentheses matter:

```
int a;
decltype(a)   // int
decltype((a)) // int&
```

- Formal parameters of functions (rule 1 applies):

```
void foo(int a, int& b, float&& c, int* d) {
  decltype(a)   // int
  decltype(b)   // int&
  decltype(c)   // float&&
  decltype(d)   // int*
  ...
}
```

- Function types (rule 1 applies):

```
int foo(char);
int bar(char);
int bar(int);
decltype(foo)       // int(char)
decltype(bar)       // error, bar is overloaded
```

  Note that rule 3 applies when a pointer to a function is formed:

```
decltype(&foo)      // int(*)(char)
decltype(*&foo)     // int(&)(char)
```

- Array types (rule 1 applies):

```
int a[10];
decltype(a);    // int[10]
```

- Member variables (member access operators):

  The type given by `decltype` is the type declared as the member variables type, so again, rule 1 applies. In particular, the cv-qualifiers originating from the *object expression* within a `.` operator or from the *pointer expression* within a `->` expression do not contribute to the declared type of the expression that refers to a member variable. Similarly, the l- or rvalueness of the object expression does not affect whether the `decltype` of a member access operator is a reference type or non-reference types.

```
class A {
int a;
int& b;
static int c;

void foo() {
  decltype(a);          // int
  decltype(this->a)     // int
  decltype((*this).a)   // int
  decltype(b);    // int&
  decltype(c);    // int (static members are treated as variables in namespace scope)
}

void bar() const {
```

```
    decltype(a);    // int
    decltype(b);    // int&
    decltype(c);    // int
  }
 ...
};

A aa;
const A& caa = aa;

decltype(aa.a)    // int
decltype(aa.b)    // int&
decltype(caa.a)   // int
```

Note that member variable names are not in scope in the class declaration scope.

```
class B {
  int a;
  enum B_enum { b };

  decltype(a) c;                          // error, a not in scope
  decltype(a) foo() { ... };              // error, a not in scope

  decltype(b) enums_are_in_scope() { return b; } // ok
  ...
};
```

Built-in operators `.*` and `->*` follow the `decltype` rule 3: l- or rvalueness of the expression determines whether the result of `decltype` is a reference or a non-reference type.

Using the classes and variables from the example above:

```
decltype(aa.*&A::a)    // int&
decltype(aa.*&A::b)    // illegal, cannot take the address of a reference member
decltype(caa.*&A::a)   // const int&
```

- `this` (rule 3 applies):

```
class X {
  void foo() {
    decltype(this)      // X*, ''this'' is ''non-lvalue'' (see 9.3.2 (1))
    decltype(*this)     // X&
  ...
  }
  void bar() const {
    decltype(this)      // const X*
    decltype(*this)     // const X&
  ...
  }
};
```

- Pointers to member variables and functions (rule 1 applies):

```
class A {
  ...
```

```
  int x;
  int& y;
  int foo(char);
  int& bar() const;
};

decltype(&A::x)        // int A::*
decltype(&A::y)        // error: pointers to reference members are disallowed (8.3.3 (3))
decltype(&A::foo)    // int  (A::*) (char)
decltype(&A::bar)    // int& (A::*) () const
```

- Literals (rule 3 applies):

  String literals are lvalues, all other literals rvalues.

```
decltype("decltype")                // const char(&)[9]
decltype(1)                         // int
```

- Redundant references (`&`) and cv-qualifiers.

  Since a `decltype` expression is considered syntactically to be a *typedef-name*, redundant cv-qualifiers and `&` specifiers are ignored:

```
int& i = ...;
const int j = ...;
decltype(i)&          // int&. The redundant & is ok
const decltype(j)     // const int. The redundant const is ok
```

- Function invocations (rule 2 applies):

```
int foo();
decltype(foo())        // int

float& bar(int);
decltype (bar(1))      // float&

class A { ... };
const A bar();
decltype (bar())       // const A

const A& bar2();
decltype (bar2())      // const A&
```

- built-in operators (rule 3 applies):

```
decltype(1+2)          // int (+ returns an rvalue)
int* p;
decltype(*p)           // int& (* returns an lvalue)
int a[10];
decltype(a[3]);        // int& ([] returns an lvalue)

int i; int& j = i;
decltype (i = 5)       // int&, because assignment to int returns an lvalue
decltype (j = 5)       // int&, because assignment to int returns an lvalue
```

```
decltype (++i);       // int&
decltype (i++);       // int (rvalue)
```

2.4 Decltype and SFINAE

If decltype is used in the return type or a parameter type of a function, and the type of the expression is dependent on template parameters, the validity of the expression cannot in general be determined before instantiating the template function. For example, before instantiating the `add` function below, it is not possible to determine whether `operator+` is defined for types `A` and `B`:

```
template <class A, class B>
void add(const A& a, const B& b, decltype(a + b)& result);
```

Obviously, calling this function with types that do not support `operator+` is an error. However, during overload resolution the function signature may have to be instantiated, but not end up being the best match, or not even be a match at all. In such a case it is less clear whether an error should result. For example:

```
template <class T, class U>
void combine(const T& t, const U& u, decltype(t + u)& result);

class A { ... };
void combine(const A& a, const A& b, std::ostream& o);

A a, b;
...
combine(a, b, cout);
```

Here, the latter `combine()` function is the best, and only, matching function. However, the former prototype must also be examined during overload resolution, in this case to find out that it is not a matching function. Argument deduction gives formal parameters `a` and `b` the type `A`, and thus the `decltype` expression is erroneous (we assume here that `operator+` is not defined for type `A`). We can identify three approaches for reacting to an operand of `decltype` which is dependent and invalid during overload resolution (by invalid we mean a call to a non-existing function or an ambiguous call, we do not mean a syntactically incorrect expression).

1. Deem the code ill-defined.

   As the example above illustrates, generic functions that match broadly, and contain decltype expressions with dependent operands in their arguments or return type, may cause calls to unrelated, less generic, or even non-generic, exactly matching functions to fail.

2. Apply the "SFINAE" (Substitution-Failure-Is-Not-An-Error) principle (see 14.8.2.). Overload resolution would proceed by first deducing the template arguments in deduced context, substituting all template arguments in non-deduced contexts, and use the types of formal function parameters that were in deduced context to resolve the types of parameters, and return type, in non-deduced context. If the substitution process leads to an invalid expression inside a decltype, the function in question is removed from the overload resolution set. In the example above, the templated `add` would be removed from the overload set, and not cause an error.

   Note that the operand of `decltype` can be an arbitrary expression. To be able to figure out its validity, the compiler may have to perform overload resolution, instantiate templates (speculatively), and, in case of erroneous instantiations back out without producing an error. To require such an ability from a compiler is problematic; there are compilers where it would be very laborious to implement.

3. Unify the rules with `sizeof` (something in between of approaches 1. and 2.)

   The problems described above are not new, but rather occur with the `sizeof` operator as well. Core issue 339: "Overload resolution in operand of sizeof in constant expression" deals with this issue. 339 suggests restricting what kind of expressions are allowed inside sizeof in template signature contexts.

The first rule is not desirable because distant unrelated parts of programs may have surprising interaction (cf. ADL). The second rule is likely not possible in short term, due to implementation costs. Hence, we suggest that the topic is bundled with the core issue 339, and rules for `sizeof` and `decltype` are unified. However, it is crucial that no restrictions are placed on what kinds of expressions are allowed inside `decltype`, and therefore also inside `sizeof`. We suggest that issue 339 is resolved to require the compiler to fail deduction (apply the SFINAE principle), and not produce an error, for as large set of invalid expressions in operands of `sizeof` or `decltype` as is possible to comfortably implement. We wish that implementors aid in classifying the kinds of expressions that should produce errors, and the kinds that should lead to failure of deduction.

3 Proposed wording

3.1 Wording for `decltype`

Section 2.11 Keywords [lex.key]

Add `decltype` to Table 3.

Section 3.2 One definition rule [basic.def.odr]

The first sentence of the Paragraph 2 should be:

> An expression is *potentially evaluated* unless it appears where an integral constant expression is required (see 5.19), is the operand of the `sizeof` operator (5.3.3) **or the `decltype` operator ([dcl.type.decltype])**, or is the operand of the `typeid` operator and the expression does not designate an lvalue of polymorphic class type (5.2.8).

Core issue 454 may change the wording slightly.

Section 4.1 Lvalue-to-rvalue conversion [conv.lval]

Paragraph 2 should read:

> The value contained in the object indicated by the lvalue is the rvalue result. When an lvalue-to-rvalue conversion occurs within the operand of `sizeof` (5.3.3) **or `decltype` ([dcl.type.decltype])** the value contained in the referenced object is not accessed, since ~~that operator does~~ **those operators do** not evaluate ~~its~~**their** operand**s**.

Section 7.1.5 Type specifiers [dcl.type]

The list of exceptions in paragraph 1 needs a new item.

> As a general rule, at most one *type-specifier* is allowed in the complete *decl-specifier-seq* of a *declaration*. The only exceptions to this rule are the following:

— `const` or `volatile` can be combined with any other type-specifier. However, redundant cv-qualifiers are prohibited except when introduced through the use of typedefs (7.1.3), **`decltype` ([dcl.type.decltype]),** or template type arguments (14.3), in which case the redundant cv-qualifiers are ignored.

Section 7.1.5.2 Simple type specifiers [dcl.type.simple]

In paragraph 1, add the following to the list of simple type specifiers:

`decltype ( ` ***expression*** **`)`**

To Table 7, add the line:

`decltype (` ***expression*** **`)`**	**the type as defined below**

Add a new paragraph after paragraph 3:

The type denoted by `decltype(e)` is defined as follows:

1. If `e` is an *id-expression* or a class member access (5.2.5 [expr.ref]), `decltype(e)` is defined as the type of the entity named by `e`. If there is no such entity, or `e` names a set of overloaded functions, the program is ill-formed.
2. If `e` is a function call (5.2.2 [expr.call]) or an invocation of an overloaded operator (parentheses around `e` are ignored), `decltype(e)` is defined as the return type of that function.
3. Otherwise, where `T` is the type of `e`, if `e` is an lvalue, `decltype(e)` is defined as `T&`, otherwise `decltype(e)` is defined as `T`.

The operand of the `decltype` operator is not evaluated.

[*Example:*

```
const int&& foo();
int i;
struct A { double x; }
const A* a = new A();
decltype(foo());   // type is const int&&
decltype(i);       // type is int
decltype(a->x);    // type is double
decltype((a->x)); // type is const double&
```

— *end example*]

Section 14.6.2.1 [temp.dep.type] Dependent types

Add a case for `decltype` in the paragraph 6:

A type is dependent if it is:

— denoted by `decltype (`*expression*`)`, where *expression* is type-dependent ([temp.dep.expr]).

Section 9.3.2 The this pointer ([class.this])

This change is not strictly necessary for `decltype`, it is a "clean-up" change, and not intended to change semantics. Paragraph 1 should start:

> In the body of a nonstatic (9.3) member function, the keyword `this` is ~~a non-lvalue~~ **an rvalue** expression
> ...

References

[Hin06] Howard Hinnant. A proposal to add an rvalue reference to the c++ language: Proposed wording, revision 2. Technical Report N1952=06-0022, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming Language C++, January 2006.

[JS03] J. Järvi and B. Stroustrup. Mechanisms for querying types of expressions: Decltype and auto revisited. Technical Report N1527=03-0110, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming Language C++, September 2003. `http://anubis.dkuug.dk/jtc1/sc22/wg21/docs/papers/2003/n1527.pdf`.

[JS04] Jaakko. Järvi and Bjarne Stroustrup. Decltype and auto (revision 3). Technical Report N1607=04-0047, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming Language C++, March 2004.

[JSGS03] Jaakko Järvi, Bjarne Stroustrup, Douglas Gregor, and Jeremy Siek. Decltype and auto. C++ standards committee document N1478=03-0061, April 2003. `http://anubis.dkuug.dk/jtc1/sc22/wg21/docs/papers/2003/n1478.pdf`.

[JSR04] Jaakko. Järvi, Bjarne Stroustrup, and Gabriel Dos Reis. Decltype and auto (revision 4). Technical Report N1705=04-0145, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming Language C++, September 2004.

[JSR06] Jaakko. Järvi, Bjarne Stroustrup, and Gabriel Dos Reis. Decltype (revision 5). Technical Report N1978=06-0048, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming Language C++, April 2006.

[Str02] Bjarne Stroustrup. Draft proposal for "typeof". C++ reflector message c++std-ext-5364, October 2002.

4 Acknowledgments

We are grateful to Jeremy Siek, Douglas Gregor, Jeremiah Willcock, Gary Powell, Mat Marcus, Daveed Vandevoorde, David Abrahams, Andreas Hommel, Peter Dimov, and Paul Mensonides, Howard Hinnant, Jens Maurer, and Jason Merrill for their valuable input in preparing this proposal.