

# Decltype (revision 5)

Programming Language C++  
Document no: N1978=06-0048

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*

2006-04-24

## 1 Introduction

We suggest extending C++ with the `decltype` operator for querying the type of an expression. Further, we suggest a new function declaration syntax, which allows one to place the return type expression syntactically after the list of function parameters.

This document is a revision of the documents 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 Lillehammer meeting, April 2004, and includes some changes to earlier specifications. [JSR04, JS04]. Most notably, the proposed wording has changed significantly, and achieves the same semantics more succinctly now.

We assume the reader is familiar with the motivation for and history behind `decltype`, and include only minimal background discussion in this document; see N1607=04-0047 [JS04] for more of the history.

## 2 The `decltype` operator

### 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. For example:

```
template<class T>
T min(const T& a, const T& b) { return a < b ? a : b; }
```

In other cases this is not as easy, or even possible. Consider the following example:

```
template<class A, class B>
R add(const A& a, const B& b) { return a + b; }
```

The return type, denoted here with `R`, should be the type of the expression `a + b`. That type may, however, be `A`, `B`, or something entirely different. In short, the type `R` depends on the types `A` and `B` in a way that is not expressible in today's C++.

Functions where the return type depends on the argument types in a non-trivial way are common in generic libraries, such as `tr1::bind`, `tr1::function`, and `Boost.Lambda`, and `Boost.uBLAS`.

## 2.2 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.

## 2.3 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 literals and as calls to built-in operators, lvalueness implies a reference type.

The semantics of the `decltype` can be described succinctly with the help of l/rvalueness of the expression as follows (these rules are directly from the proposed wording given in Section 5):

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

1. If `e` is of the form `(e1)`, `decltype(e)` is defined as `decltype(e1)`.
2. If `e` is a name of a variable or non-overloaded function, `decltype(e)` is defined as the type used in the declaration of that variable or function. If `e` is a name of an overloaded function, the program is ill-formed.
3. If `e` is an invocation of a user-defined function or operator, `decltype(e)` is the return type of that function or operation.
4. Otherwise, where `T` is the type of `e`, if `T` is `void` or `e` is an rvalue, `decltype(e)` is defined as `T`, otherwise `decltype(e)` is defined as `T&`.

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

## 2.4 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)
```

### 2.4.1 Variable and function names

Situations where the second bullet in Section 2.3 applies:

- Variables in namespace or local scope:

```

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

```

- Formal parameters of functions:

```

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

```

- Function types:

```

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

```

Note that rule 4 applies when a pointer to a function is formed:

```

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

```

- Array types:

```

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

```

- Member variables:

The type given by `decltype` is the type declared as the member variables type. 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

```

Handling references to member variables has changed from the rules proposed in N1705=04-0145 [JSR04], back towards the specification in 1607=04-0047 [JS04]. The N1705=04-0145 specification considered accesses to member variables as member access operators, whose type was defined using the rule 4. The current specification retains more information on how the members have been declared — `decltype` in the N1705=04-0145 specification could not distinguish whether a member was defined as a reference or a non-reference type.

Member variable names are not in scope in the class declaration scope. Should this be seen as a serious restriction, relaxing it ought to be considered.

```

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

    decltype(a) c;           // error, a not in scope
    static const int x = sizeof(a); // error, a not in scope

    decltype(this->a) c2;     // error, this not in scope
    decltype((B*)0->a) hack; // error, B* is incomplete

    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 4: l- or rvalue-ness 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`:

```
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:

```
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:

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:

```
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

```

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)

```

## 2.5 decltype and forwarding functions

*Forwarding functions* wrap calls to other functions. They are functions that forward their parameters, or some expressions computed from them, into another function and return the result of that function. In the case of generic forwarding functions, return type of the forwarding function may depend on the types of its actual arguments: the function called from the forwarding function can be overloaded, or a template. For such a forwarding function to be transparent, its return type should match exactly with the return type of the wrapped function, no matter with what types the forwarding function template is instantiated. It is not in general possible to write type expressions that would accomplish this in today's C++; providing this ability is one of the main motivations for `decltype`.

The key property of `decltype` for enabling generic forwarding functions is that no essential information on whether a function returns a reference type or not, is lost. The following example demonstrates why this is crucial:

```

int& foo(int& i);
float foo(float& f);

template <class T> void only_lvalues(T& t) { ... }; // doesn't accept temporaries

template <class T> auto transparent_forwarder(T& t) -> decltype(foo(t)) {
    ...; return foo(t);
}

int i; float f;
only_lvalues(foo(i)); // ok
only_lvalues(transparent_forwarder(i)); // should be ok too

only_lvalues(foo(f)); // not ok
only_lvalues(transparent_forwarder(f)); // should not be ok either

```

Further, similar forwarding should work with built-in operators:

```

template <class T, class U>
auto forward_foo_to_comma(T& t, U& u) -> decltype(foo(t), foo(u)) {
    return foo(t), foo(u);
}

int i; float f;
forward_foo_to_comma(i, f); // should return float
forward_foo_to_comma(f, i); // should return int&

```

This behavior would be easily attained with a an operator that results in a reference type if and only if its operand is an lvalue (see Section 4 for a discussion on an alternative proposal for the `decltype` operator with this semantics). The proposed `decltype` operator obeys this rule except for member variable accesses. This deviation from the rule is not serious. Member variable accesses are syntactically distinct from any other forms of expression, and it is not difficult to attain lvalue/rvalue semantics for the case where member access is the expression being forwarded.

For example, when calling the following forwarding function with argument `a`, the expression `t.data` is an lvalue, but `decltype(t.data)`, and thus `forward_data` function, has return the type `int`.

```

class A { int data; }

template <class T>
auto forward_data (T& t) -> decltype(t.data) { return t.data; }

A a;
forward_data(a); // rvalue

```

Rewriting the `forward_data()` function as

```

template <class T>
auto forward_data (T& t) -> decltype(void(), t.data) { return t.data; }

```

The comma operator returns `t.data`, but the result type is now computed with `decltype`'s rule 4, and thus is a reference type. To guarantee the use of built-in comma operator, the left-hand side is a `void` operation.

## 2.6 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.

Note that this option gives programmers the power to query (at compile time) whether a type, or sequence of types, support a particular operation. One can also group a set of operations into one `decltype` expression, and test its validity (cf. *concepts*). It would also be possible to overload functions based on the set of operations that are valid: For example:

```
template <class T>
auto advance(T& t, int n) -> decltype(t + n, void()) {
    t + n;
}
```

This function would exist only for such types `T`, for which `+` operation with `int` is defined.

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 New function declaration syntax

We anticipate that a common use for the `decltype` operator will be to specify return types that depend on the types of function arguments. Unless the function's argument names are in scope in the return type expression, this task becomes unnecessarily complicated. For example:

```
template <class T, class U> decltype((*T*)0)+(*U*0) add(T t, U u);
```

The expression `(*T*)0` is a hackish way to write an expression that has the type `T` and does not require `T` to be default constructible. If the argument names were in scope, the above declaration could be written as:

```
template <class T, class U> decltype(t+u) add(T t, U u);
```

Several syntaxes that move the return type expression after the argument list are discussed in [Str02]. If the return type expression comes before the argument list, parsing becomes difficult and name lookup may be less intuitive; the argument names may have other uses in an outer scope at the site of the function declaration.

We suggest reusing the `auto` keyword to express that the return type is to follow after the argument list. The return type expression is preceded by `->` symbol, and comes after the argument list and potential cv-qualifiers in member functions and the exception specification:

```
template <class T, class U> auto add(T t, U u) -> decltype(t + u);
class A {
    auto f() const throw () -> int ;
};
```

The syntax with which a function is declared is insignificant. For example, the following two function declarations declare the same function:

```
auto foo(int) -> int;
int foo(int);
```

The new syntax should be allowed to express function types in any context where a function type is allowed today. We see this as a natural opportunity to introduce a more readable syntax for types that involve function types, which are a source of unnecessary difficulty in today's C++.

We do not present the wording for the new function declaration syntax in this proposal. What we are proposing here differs from what was discussed in the Berlin meeting, and all details on how the syntax fits into existing wording in the standard, and whether it is free of parsing problems, have not yet been fully worked out. Instead, we present the ideas informally.

The new function type syntax should be allowed as such in defining and declaring functions. In all other uses, the function type must be enclosed in parentheses, and it behaves syntactically as a *typedef-name* (except for the rules of eliminating redundant cv-qualifiers and `&` specifiers). The following examples illustrate:

```
auto f(int)->int { ... }           // ok, function definition
auto f(int)->int;                  // ok, function declaration

typedef auto F(int)->int;          // error
typedef (auto (int)->int) F;      // ok

typedef auto (*PF)(int)->int;     // error
typedef (auto (int)->int)* PF;    // ok, pointer to function

(auto (int)->int)* a[10];         // ok, array of function pointers
```

To demonstrate the improved readability, consider the two function declarations, both declaring the same function prototype taking a pointer to a function from `int` to `void` as its second parameter, and returning a pointer to a function of the same type:

```
void (*signal(int signum, void (*handler)(int)))(int);

auto signal( int signum, (auto (int) -> void)* handler) -> (auto (int) -> void)*
```

Of course, typedefs can be used to improve the readability of complex type expressions involving function types.

## 4 The valtype proposal

Jason Merrill suggested in a draft proposal D1896 that `decltype` should be named `valtype`, and its semantics changed so that the l/rvalueness of the operand of `valtype` determines in all cases whether the operator gives a reference type or not. Basically semantics of `valtype` would be obtained by removing rule 2 in the numbered list of rules in Section 2.3.

The `valtype` semantics are slightly simpler to specify, and possibly a bit easier to implement, but no difficulties is expected in implementing either semantics. We do not feel very strongly which semantics to use: both are correct for defining forwarding functions, which is the main motivation for introducing `decltype`, or a similar operator, to C++. We do think, however, that `valtype` is a misleading name. We easily interpret the name to suggest giving a “value type” where a possible reference has been removed.

The `valtype` semantics were discussed both in the Santa Cruz and Oxford meetings (also advocated by some of the authors of this proposal) but did not get support in the EWG. One of the examples that was considered troublesome with that semantics was the following (though the name `valtype` was not mentioned in those discussions):

```
int i;
valtype(i) j = f(); //j would be of type int&
```

The `valtype` proposal argues that the above is not a real use case, and the code would instead likely be written as follows:

```
int i;
auto j = f();
```

We do not disagree with this observation.

The benefits of the `valtype` semantics over the `decltype` semantics include a small gain in simplicity of the specification, possibly implementation. Moreover, l- and r-valueness would become more directly part of the C++ type system by classifying l-values to be of reference types, and r-values of non-reference types. On the other hand, there are cases where the `valtype` semantics may be surprising. A slight variation of the example above is where one or more uninitialized variables are declared to have the same type than some other variable. Uninitialized variables are useful, for example as “output arguments” of functions. In the code below, `tmp1` and `tmp2` would get a reference type with the `valtype` semantics, and render the code ill-formed:

```
auto i = f();
valtype(i) tmp1, tmp2;
```

Another case of declaring variables without an initializer are extern variables. The following example is taken from the GNU libc library (where we have replaced the uses of the non-standard `__typeof__` with the `valtype` operator):

```
extern valtype (uselocale) __uselocale;
/* ... */
extern valtype (wcstol_l) __wcstol_l;
```

This would declare the above variables to have reference types, which is not the intention. To get the right semantics, for example the first line would have to be written as:

```
extern std::remove_reference<valtype (uselocale)>::type __uselocale;
```

or alternatively, using `decltype`:

```
extern decltype (uselocale) __uselocale;
```

The `decltype` of a member variable is always a reference type, and thus the information whether a member variable was declared to be a reference or non-reference type is lost. As discussed in Section 2.4.1, `decltype` retains this information. We do not know whether there are important use cases that would benefit from this ability.

One possible use of the `decltype` or `decltype` operator would be to merely avoid typing long type names:

```
auto foo(a_very_long_argument_type_too_long_to_write_twice a) -> decltype(a);
```

With `decltype` semantics, this function's return type would be a reference type.

In sum, either the `decltype` or `decltype` semantics solve the forwarding problem, which is the main motivation for a `decltype`-like operator. The arguments for choosing one over the other are not strikingly strong to either direction.

Finally, note that the following macro defines `decltype` in terms of `decltype`:

```
#define decltype(e) decltype(void(),e)
```

## 5 Proposed wording

### 5.1 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** operands.

#### 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` ([`decl.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 [`decl.type.simple`]

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

`decltype ( expression )`

To Table 7, add the line:

<code>decltype ( expression )</code>	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 of the form `(e1)`, `decltype (e)` is defined as `decltype (e1)`.
2. If `e` is a name of a variable or non-overloaded function, `decltype (e)` is defined as the type used in the declaration of that variable or function. If `e` is a name of an overloaded function, the program is ill-formed.
3. If `e` is an invocation of a user-defined function or operator, `decltype (e)` is the return type of that function or operation.
4. Otherwise, where `T` is the type of `e`, if `T` is `void` or `e` is an rvalue, `decltype (e)` is defined as `T`, otherwise `decltype (e)` is defined as `T&`.

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

### 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`])

Paragraph 1 should start:

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

## 5.2 New function declaration syntax that moves the return type expression after the parameter list

Wording not yet provided, as explained in Section 3 of this proposal.

## References

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

## 6 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 for their valuable input in preparing this proposal. Clearly, this proposal builds on input from members of the EWG as expressed in face-to-face meetings and reflector messages. Jens Maurer contributed to the proposed wording in this document.