Decltype and auto (revision 4)

Programming Language C++

Document no: N1705=04-0145

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

September 12, 2004

1 Background

This document is a revision of the documents N1607=04-0047 [JS04], N1527=03-0110 [JS03], and N1478=03-0061 [JSGS03], and builds also on [Str02]. We assume the reader is familiar with the motivation for and history behind ***decltype*** and ***auto***, and do not include background discussion in this document; see N1607=04-0047 [JS04] for that. We merely summarize, with two examples, the key capabilities that the proposed features would bring to the language:

1. Declaring return types of functions that depend on the parameter types in a non-trivial way:

```
template <class A, class B>
auto operator+(const Matrix<A>& a, const Matrix<B>& b) -> Matrix<decltype(a(0, 0) + b(0, 0))>;
```

2. Deducing the type of a variable from the type of its initializer:

```
template <class A, class B>
void foo(const A& a, const B& b) {
    auto tmp = a * b;
    ...
```

1.1 Changes from N1607

Changes to the previous revisions primarily reflect the EWG discussions and results of the straw votes that took place in the Kona and Sydney meetings. In addition, we suggest a change to the rules of ***decltype*** for certain built-in operators, and for expressions that refer to member variables. Following is the list of changes from proposal N1607.

- The decltype rules now explicitly state that ***decltype((e)) == decltype(e)*** (as suggested by EWG).
- Rules for ***decltype*** of member variables have changed.
- Rules for ***decltype*** of built-in comma, pre- and postincrement, and all assignment operators have changed.

- Rules for the use of ***auto*** to declare variables have changed to allow only one variable declaration in a declarator list when the type of the variable is deduced from its initializer expression (as suggested by EWG).
- Currently allowed uses of ***auto*** will not be invalidated with the new rules (as suggested by EWG).
- Added discussion on ***decltype*** and SFINAE.
- Removed wording for *implicit templates* using ***auto*** (as suggested by EWG).
- Removed wording for using ***auto*** to mean that the return type of a function should be deduced from the expression in the ***return*** statement of the function (as suggested by EWG).
- Removed the section describing changes between N1607 and N1478, and section on background and history.

2 Introduction

We suggest extending C++ with the ***decltype*** operator for querying the type of an expression, and allowing the use of keyword ***auto*** to indicate that the compiler should deduce the type of a variable from its initializer 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. Previous revisions of this proposal explored the possibility of using the ***auto*** keyword to denote implicit template parameters, and to instruct the compiler to deduce the return type of a function from its body. Neither of these features gained significant support in EWG, and thus are not brought forward at this point.

3 The *decltype* operator

Guiding principle: The rules of determining the type ***decltype(e)*** build on a single guiding principle: look for the declared type of the outermost expression in ***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 read the result of ***decltype*** directly from the program text. In addition to this principle, there are rules for expressions where the outermost node does not have a declaration in the program text, such as built-in operators, and literals.

Syntax: The syntax of ***decltype*** is:

simple−type−specifier
  ...
  decltype (expression)
  ...

We require parentheses (as opposed to ***sizeof***'s more liberal rule) to keep the syntax simple and to keep the door open for inquiry operations on the results of ***decltype***, e.g. ***decltype(e).is_reference()***. We do not, however, propose any such extensions at this point. Syntactically, ***decltype(e)*** is treated as if it were a *typedef-name* (cf. 7.1.3). The semantics of the ***decltype*** operator is described with the following rules (note that the content of the next subsection 3.1 is part of the suggested standard wording):

3.1 Decltype rules

The type denoted by a decltype type expression ***decltype(e)*** is the declared type of the outermost expression node of its argument ***e***, defined as:

1. If ***e*** is of the form ***(e1)***, ***decltype(e)*** is defined as ***decltype(e1)***.

2. If ***e*** is a name of a variable in namespace or local scope, a static member variable, a formal parameter of a function, or a non-overloaded name of a function, ***decltype(e)*** is the declared type of that variable, formal parameter, or function. In particular, ***decltype(e)*** results in a reference type if, and only if, the variable or formal parameter is declared to have a reference type. 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 declared return type of that function or operator.
4. If ***e*** refers to a member variable, ***e*** is transformed to a member access operator (see 5.1. (7)), after which rule 5 is applied.
5. If ***e*** is an invocation of a built-in operator and has one of the following forms:

   e1, e2
   ++e1
   −−e1
   e1@e2

   where **@** stands for any assignment operator, then ***decltype(e)*** is computed as follows:

decltype(e1,e2)	is defined as	***decltype(e2)***
decltype(++e1)	is defined as	***decltype(e1)***
decltype(−−e1)	is defined as	***decltype(e1)***
decltype(e1@e2)	is defined as	***decltype(e1)***

   Let ***T*** be the expression type of ***e***. If ***e*** is an invocation of any other built-in operator, and if ***e*** is an rvalue, ***decltype(e)*** is ***T***. If ***e*** is an invocation of any other built-in operator, and if ***e*** is an lvalue, ***decltype(e)*** is ***T&***.
6. If ***e*** is an rvalue literal ([expr.prim]) of type ***T***, then ***decltype(e)*** is the non-reference type ***T***. If ***e*** is an lvalue literal ([expr.prim]) of type ***T***, then ***decltype(e)*** is the reference type ***T&***.

The operand of a ***decltype*** expression is not evaluated. A ***decltype*** expression that would result in an unnamed type is ill-formed. Syntactically, a ***decltype*** type expression is treated as if it were a *typedef-name* (cf. 7.1.3).

3.2 Decltype examples and discussion

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)
```

In the following we give examples of ***decltype*** with different kinds of expressions:

- 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&
```

- 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*
  ...
}
```

- 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 decltype(i) is int
decltype (j = 5) // int&, because decltype(j) is int&

decltype (++i); // int, because decltype(i) is int
decltype (++j); // int&, because decltype(j) is int&
decltype (i++); // int (rvalue)
decltype (j++); // int (rvalue)
```

- Function types:

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

- Array types:

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

- Pointers to member variables and member 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
```

- Member variables:

  The type given by ***decltype*** is the type of the member access operation that the expression denotes. In particular, the cv-qualifiers originating from the *object expression* within a **.** operator or from the *pointer expression* within a −> expression contribute to the declared type of the expression that refers to a member variable. Similarly, the l- or rvalueness of the object expression affects the l- or rvalueness of the member access operator, and is reflected in the type given by ***decltype***.

  Note, that this is a change from the rules proposed in N1607, where the cv-qualifiers, or l- or rvalueness did not contribute to the result of decltype. Neither choice is perfect, but we find the currently proposed rule less confusing. In particular, with the previous rules the decltype operator could give different results for expressions that access the same member using the **.*** operator or using the member access operator. The same was true with −>* and −> operators. Now it is guaranteed that both mechanisms yield the same result. Philosophically, accessing a member is an operation where the object expression affects the outcome, rather than just a lookup of an independent name. Therefore, ***decltype*** should follow the rules for operations. As a downside of the changed rules, one cannot (at least not easily) distinguish between whether a member was declared to be of a reference type, or of a non-reference type. This information may be useful to the programmer, but the mechanism for that should rather be provided separately. For example, we could allow one to write ***decltype(A::x)*** to express the intent of querying the declared type of the member ***x*** of class ***A*** without a reference to a particular object.

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

 void foo() {
  decltype(a);   // int& (member access operator returns an lvalue)
  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);   // const int&
  decltype(b);   // int&
  decltype(c);   // int
 }
 ...
};

A aa;
const A& caa = aa;

decltype(aa.a)    // int&
decltype(aa.b)    // int&
decltype(caa.a)  // const 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
 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
 fun bar() -> decltype(a) { ... }; // still an error

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

Should this be seen as a serious restriction, we can consider relaxing it, but we see no current need for that.

Built-in operators `.*` and `->*` follow the ***decltype*** rule 4: l- or rvalueness of the expression determines whether decltype give 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*
  decltype(*this)   // X&
 ...
 }
 void bar() const {
  decltype(this)    // const X*
  decltype(*this)   // const X&
```

```
  ...
 }
};
```

- 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
```

3.3 *decltype* and forwarding functions

Forwarding functions are simply functions that wrap calls to other functions. One particular class of such functions forwards its arguments, or results of expressions containing some of these arguments, to some other function, and returns the result of this invocation. 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 is easily attained with a full "reference-preserving" typeof operator with just one rule: if the expression whose type is being examined is an lvalue, the resulting type should be a reference type; otherwise, the resulting type should not be a reference type. The proposed ***decltype*** operator obeys this rule except for a handful of built-in operators. The deviation from the rule has minimal effect on forwarding functions. In particular, if all arguments that are used in a decltype expression that defines the return type of the forwarding function are of reference types, the deviation has no impact.

The reasoning behind introducing the special rules for the few built-in operators is to better reflect the behavior of the operators. For example:

```
template <class T, class U>
auto forward_comma(T t, U u) -> decltype(t, u) {
  return t, u;
}

int i; float f;
forward_comma(i, f); // float
```

The expression ***(t, u)*** is an lvalue. Was ***decltype(t, u)*** defined to result an lvalue, the ***forward_comma*** function would be in error (trying to return a reference to a local variable). This is avoided with the rule that unifies the decltype of a built-in comma operator with the decltype of its right-hand argument. Similar reasoning is behind the rules for built-in assignment operators, and prefix increment and decrement operators.

The rule for the built-in comma operator, prefix increment and decrement operators, and all built-in assignment operators, may in some cases require examining more than just the topmost expression node to decide what the ***decltype*** of an expression is. It is not enough to know the topmost node and the types of its arguments; the compiler needs to know the *declared types* of the arguments:

```
int a, b, c, d; int& e = d;
decltype(a, (b, (c, d))); // int
decltype(a, (b, (c, e))); // int&
```

Here, the declared type of the leaf node determines the declared type of the whole expression.

3.4 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. In such a case it is less clear whether an error should result. For example:

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

```
void add(int*& res, int* p, int n);

int* p = new int[100];
int* result;
int steps = 10;
add(result, p, steps);
```

Here, the latter overload is the best matching function. However, the former prototype must also be examined during overload resolution. Argument deduction gives formal parameters ***a*** and ***b*** pointer types, for which ***operator+*** is not defined. We can identify three approaches for reacting to a operand of ***decltype*** which is dependent and invalid (e.g. a call to a non-existing function or an ambiguous call, not syntactically incorrect) during overload resolution:

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 caused worries among implementors during the discussion of the ***concept*** proposals in the Kona meeting.

   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.

4 Auto

We suggest that the ***auto*** keyword would indicate that the type of a variable is to be deduced from its initializer expression [Str02]. For example:

```
auto x = 3.14; // x has type double
```

The semantics of ***auto*** should follow exactly the rules of template argument deduction. The ***auto*** keyword can occur in any deduced context in a type expression. Examples (the notation ***x : T*** is read as "***x*** has type ***T***"):

```
int foo();
auto x1 = foo();              // x1 : int
const auto& x2 = foo();   // x2 : const int&
auto& x3 = foo();           // x3 : int&: error, cannot bind a reference to a temporary

float& bar();
auto y1 = bar();              // y1 : float
const auto& y2 = bar();   // y2 : const float&
auto& y3 = bar();           // y3 : float&

A* fii()
auto* z1 = fii();           // z1 : A*
auto z2 = fii();            // z2 : A*
auto* z3 = bar();           // error, bar does not return a pointer type
```

A major concern in discussions of ***auto***-like features has been the potential difficulty in figuring out whether the declared variable will be of a reference type or not. Particularly, is unintentional aliasing or slicing of objects likely? For example

```
class B { ... virtual void f();  }
class D : public B { ... void f(); }
B* d = new D();
...
auto b = *d;   // is this casting a reference to a base or slicing an object?
b.f();             // is polymorphic behavior preserved?
```

Basing ***auto*** on template argument deduction rules provides a natural way for a programmer to express his intention. Controlling copying and referencing is essentially the same as with variables whose types are declared explicitly. For example:

```
A foo();
A& bar();
...
A x1 = foo();       // x1 : A
auto x1 = foo();    // x1 : A

A& x2 = foo();       // error, we cannot bind a non-lvalue to a non-const reference
auto& x2 = foo();   // error

A y1 = bar();       // y1 : A
auto y1 = bar();    // y1 : A

A& y2 = bar();       // y2 : A&
auto& y2 = bar();   // y2 : A&
```

Thus, as in the rest of the language, value semantics is the default, and reference semantics is provided through consistent use of ***&***. The type deduction rules extend naturally to more complex definitions:

```
std::vector<auto> x = foo();
std::pair<auto, auto>& y = bar();
```

The declaration of ***x*** would fail at compile time if the return type of ***foo*** was not an instance of ***std::vector***, or a type that derives from an instance of ***std::vector***. Analogously, the return type of ***bar*** must be an instance of ***std::pair***, or a type deriving from such an instance. Declaring such partial types for variables can be seen as documenting the intent of the programmer. Here, the compiler can enforce that the intent is satisfied.

The straw votes from Sydney indicated some opposition against the more complex uses of ***auto***, that is, against allowing the use of auto as a placeholder for any part of type. Partially this was because of fears of complicating implementations. Based on our discussions with some implementors the step from allowing the use of ***auto*** as a basic type specifier (allow to be used with cv-qualifiers, $*$ and ***&***) to the 'use ***auto*** as any part of type', would not be huge. Thus, we still keep this as part of the proposed features.

The suggested syntax does not allow expressing constraints between two different uses of ***auto***, e.g., requiring that both arguments to ***pair*** in the above example are the same. No feature to allow expressing such constraints is proposed at this point.

4.1 Direct initialization syntax

Direct initialization syntax is allowed and is equivalent to copy initialization. For example:

```
auto x = 1; // x : int
auto x(1); // x : int
```

The semantics of a direct-initialization expression of the form ***T v(x)*** with ***T*** a type expression containing one or more uses of ***auto***, ***v*** as a variable name, and ***x*** an expression, is defined as a translation to the corresponding copy initialization expression ***T v = x***. Examples:

```
const auto& y(x) -> const auto& y = x;
std::pair<auto, auto> p(bar()) -> std::pair<auto, auto> p = bar();
```

It follows that the direct initialization syntax is allowed with ***new*** expressions as well:

```
new auto(1);
```

The expression ***auto(1)*** has type ***int***, and thus ***new auto(1)*** has type ***int****. Combining a ***new*** expression using ***auto*** with an ***auto*** variable declaration gives:

```
auto* x = new auto(1);
```

Here, ***new auto(1)*** has type ***int****, which will be the type of ***x*** too.

5 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 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) but before the exception specification:

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

We refer to [Str02] for further analysis on the effects of the new function declaration syntax.

More examples:

```
auto f(int i) -> int;
template <class T>
auto id(T& a) -> decltype(a);
```

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);
```

6 Conclusions

In C++2003, it is not possible to express the return type of a function template in all cases. Furthermore, expressions involving calls to function templates commonly have very complicated types, which are practically impossible to write by hand. Hence, it is often not feasible to declare variables for storing the results of such expressions. This proposal describes ***decltype*** and ***auto***, two closely related language extensions that solve these problems. Intuitively, the ***decltype*** operator returns the declared type of an expression. For variables and parameters, this is the type the programmer finds in the program text. For functions, the declared type is the return type of the definition of the outermost function called within the expression, which can also be traced down and read from the program text (or in the standard in the case of built-in functions). The semantics of ***auto*** is unified with template argument deduction.

7 Proposed wording

7.1 `decltype`: proposed text

Section 2.11 Keywords.

Add `decltype` to Table 3.

Section 3.2 One definition rule

To paragraph 2, add:

> is the operand of the `decltype` operator ([dcl.type.decltype])

as one of the exceptions for *potentially evaluated.*

Section 4.1 Lvalue-to-rvalue conversion

To paragraph 2, add a case for `decltype`:

> ... 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 those operators do not evaluate their operands.

Section 7.1.5 Type specifiers

In paragraph 1:

— `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 Type specifiers

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

`decltype (` *expression* `)`

To Table 7, add the line:

`decltype (` *expression* `)`	`the declared type of the outermost expression node of` *expression*

New subsection: Decltype [dcl.type.decltype]

Should be placed after 7.1.5.2. The text of the section is the decltype rules in Section 3.1 of this document.

Section 14.6.2.1 [temp.dep.type] Dependent types

Add a case for `decltype`:

— obtained with `decltype(`*expression*`)`, where *expression* is a type-dependent expression ([temp.dep.expr]).

7.2 `auto` in variable declarations: proposed text

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

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

`auto`

To Table 7, add the line:

`auto`	placeholder for a type

Add to the paragraph following Table 7:

> The `auto` type specifier ([dcl.type.auto]) is only allowed in a *decl-specifier-sequence* that is followed by an *init-declarator-list* with exactly one *init-declarator*, which consists of a *declarator* and a non-empty *initializer*. The initializer must be of either of the following two forms:

```
= initializer−clause
( initializer−clause )
```

[*Example:* The following are valid declarations:

```
auto x = 5;
auto *v = expr;
pair<int, auto> a = pair<int, int>();
pair<int, auto> *b = new pair<int, float>();
```

— *end example*]

Section 8.3 Meaning of declarators [dcl.meaning]

New paragraph after paragraph 1:

> The *decl-specifier-sequence* of a declaration may contain one or more occurrences of the `auto` keyword if the declarator in the declaration declares an object and specifies an initial value. In this case, the type of each declared identifier is deduced from the type of its initializer ([dcl.auto]).

Replace paragraph 4 with:

> First, the `decl-specifier-seq` determines a type; or, when it contains occurrences of `auto`, a *type scheme*. A type scheme yields a type if each occurrence of `auto` in the type scheme is replaced by a type. In a declaration
>
> ```
> T D
> ```
>
> the *decl-specifier-seq* `T` determines the type, or type scheme, "`T`". [*Example:* in the declarations
>
> ```
> int unsigned i;
> pair<auto, auto> p = f();
> ```
>
> the type specifiers `int unsigned` determine the type "`unsigned int`", and the type specifier `pair<auto, auto>` determines the type scheme "`pair<auto, auto>`" ([dcl.type.simple]).]

Sections 8.3.1–6 discuss how `*`, reference, array etc. in the declarator propagate to the type of the *declarator-id*. These must be adapted to apply to type schemes in addition to types. Details not shown.

New subsection: Auto [dcl.auto]

The section should be a subsection of Section 8.3 ([dcl.meaning]). The text of the new subsection:

> Once the type scheme of a *declarator-id* has been determined, the type of each variable using the *declarator-id* is determined from the type of its initializer using the rules for template argument deduction ([temp.deduct]). Let `T` be the type scheme that has been determined for a variable identifier `d`, and `e` be the initializer expression for `d`. Obtain `T'` from `T` by replacing each occurrence of `auto` with a new unique identifier. Denote these identifiers as $t_1, \ldots, t_n$. Define a function template as follows:
>
> ```
> template <class t1, ..., class tn>
> void __f(T' __d) {}
> ```

The type deduced for the variable `d` is then the type that would be deduced for the parameter `__d` in a call to `__f` with `e` as its actual argument. If the function argument deduction would fail, the declaration is ill-formed.

[*Example:*

```
vector<auto, auto> &i = expr;
```

The type scheme is `vector<auto, auto>&`, and the type of `i` is the deduced type of the argument `__i` in the call `__f(expr)` of the following function template:

```
template <__T1, __T2> void __f(vector<__T1, __T2>& i);
```

— end example]

Section 8.5 Initializers [dcl.init]

To paragraph 14 add a case:

If the destination type contains the `auto` specifier, see section [dcl.init.auto].

Section 5.3.4 New [expr.new]

Paragraph 1 specifies the valid forms of new expressions. Add the following form for *new-type-id* to the grammar:

new−type−id:
...
cv `auto` *direct−new−declarator*$_{opt}$

And the text:

If *new-type-id* is of the form "*cv* `auto` *direct-new-declarator*$_{opt}$", *new-initializer* with exactly one initializer argument must follow *new-type-id*, or the program is ill-formed. The allocated type is deduced from the type of this initializer argument as follows: Let `(e)` be the *new-initializer*, then the allocated type is the type deduced for the variable `x` in the declaration ([dcl.auto]):

cv `auto x = e`

Once the allocated type has been deduced, the semantics of the *new-expression* is as if the form "*cv* `auto` *direct-new-declarator*$_{opt}$" was written "`T` *direct-new-declarator*$_{opt}$", where `T` is the type deduced for the allocated type. [*Example:*

```
new auto(1);        // allocated type is int
double& foo();
new const auto[10](foo()); // allocated type is const double
auto x = new auto('a'); // allocated type is char, x is of type char*
```

— end example]

7.3 New function declaration syntax that moves the return type expression after parameter list: proposed text

Section 8.3.5 Functions ([dcl.fct])

Add a new paragraph after paragraph 1:

In a declaration `auto D`, where `D` has the form

`D1 (` *parameter−declaration−clause* `)` *cv−qualifier−seq*$_{opt}$ `->` *type−id* *exception−specification*$_{opt}$

and the type scheme of the contained *declarator-id* in the declaration `auto D1` is "*derived-declarator-type-list* `auto`", the type of the *declarator-id* in `D` is "*derived-declarator-type-list* function of (*parameter-declaration-clause*) *cv-qualifier-seq*$_{opt}$ *exception-specification*$_{opt}$ returning *type-id*"; a type of this form is a *function type*.

Section 8.4 Function definitions ([dcl.fct.def])

To paragraph 1, add the new syntax as an allowed *declarator* form in function definitions. The end of the paragraph should read:

The *declarator* in a *function-definition* shall have one of the forms:

`D1 (` *parameter−declaration−clause* `)` *cv−qualifier−seq*$_{opt}$ *exception−specification*$_{opt}$
`D1 (` *parameter−declaration−clause* `)` *cv−qualifier−seq*$_{opt}$ `->` *type−id* *exception−specification*$_{opt}$

as described in 8.3.5. A function shall be defined only in namespace or class scope.

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`.

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

8 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.