Decltype and auto (revision 3)

Programming Language C++

Document no: N1607=04-0047

Jaakko Järvi
Indiana University
Pervasive Technology Laboratories
Bloomington, IN
jajarvi@osl.iu.edu

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

February 17, 2004

Contents

1 Background

This document is a revision of the document N1478=03-0061 [JS03], clarifying which of the features described in N1478=03-0061 we propose to be included into the next revision of the standard. The clarifications reflect the EWG discussions and results of the straw votes that took place in the Kona meeting. In addition, this document suggests standard wording for the proposed features.

Section 1.1 describes the changes between the document N1478 and the current proposal. Section 8 contains the proposed standard wording. The proposal describes several related but distinct features (***decltype***, and the use of ***auto*** in several contexts), thus Section 8 is divided into sections according to these individual features. Each section describes the changes and additions necessitated by a particular feature.

1.1 Changes from N1478

- N1478 suggested *implicit templates* to give a less verbose syntax for defining template functions. Support for this feature was mild in the Kona meeting, and thus we do not suggest the feature to be included at this point. For future consideration and for the purpose of documentation, however, this document includes the description and suggested standardeze for implicit templates.

- N1478 mentioned allowing template specializations for variable declarations. For example, the variable declaration:

  template <class T> std::pair<T, T> z = bar();

  would succeed as long as the result type of ***bar()*** would match ***std::pair<T, T>***. Allowing template syntax for variable declarations is not part of the current proposal. We note however, that the feature would provide a convenient mechanism for ascertaining that the type of an expression has an expected form.

- N1478 suggested adding two new syntaxes for function declarations.

  auto function−name(parameter−list) −> expression
  auto function−name(parameter−list) −> type

  The syntax with an expression following −> is not part of this proposal. First, it is not strictly necessary, because the same effect can be attained with the latter syntax by wrapping the expression in a ***decltype*** and using simple metafunctions. Second, the syntax may lead to ambiguous parses. For example:

  float f(int);
  auto (∗p)(int) −> f(0)

  What is the return type of ∗***p***? ***float***, or ***float (&)(int)***.

- Regarding implicit return types (return types deduced from the ***return*** statement in the function body) we suggested the following rule, banning the use of a function definition with an implicit return type to provide a definition for a declaration that did specify a return type.

  > If a function is declared first, any subsequent declarations, and the definition of the function must specify the return type (using any function declaration syntax), and the return type must be the same as in the first declaration. The actual expression specifying the return type can be different, though.

  This rule is a safety measure, but also a special case, and thus may not be justified. We have not included the specification of this rule in the suggested standard wording.

- ***decltype*** of string literals and functions yield a reference type, unlike in N1478.

  There is no obvious choice for either way (reference or non-reference type); the current rules are more consistent with the ***decltype*** rule for function and operator call expressions.

2 Introduction

C++ does not have a mechanism for directly querying the type of an expression. Neither is there a mechanism for initializing a variable without explicitly stating its type. Stroustrup suggests [Str02] that the language be extended with mechanisms for both these tasks, discussing broadly several different possibilities for the syntax and semantics of these mechanisms. Subsequent proposals [JSGS03, JS03] defined the exact semantics and suggested syntax for these mechanisms: the ***decltype*** operator for querying the type of an expression, and the keyword ***auto*** for indicating that the compiler should deduce the type of a variable from its initializer expression. Furthermore, the proposals 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. This proposal builds on those previous proposals and to make the proposal self-contained, we include background material from [JSGS03] and [JS03], summarizing earlier discussions on ***typeof***. In what follows, we use the operator name ***typeof*** when referring to the mechanism for querying a type of an expression in general. The ***decltype*** operator refers to the proposed variant of ***typeof***.

2.1 Motivation

C++ would benefit from ***typeof*** and ***auto*** in many ways. These features would be convenient in several situations, and increase code readability. More importantly, the lack of a ***typeof*** operator is worse than an inconvenience for many generic library authors: it is often not possible to express the return type of a generic function. This leads to hacks, workarounds, and reduced functionality with an additional burden imposed on the library user (see for example the return type deduction mechanisms in [JPL03,Dim01,WK02,Vel], or the function object classes in the standard library). Below we describe typical cases which would benefit from ***typeof*** or ***auto***. For additional examples, see [Str02].

- The return type of a function template can depend on the types of the arguments. It is currently not possible to express such return types in all cases. Many forwarding functions suffer from this problem. For example, in the ***trace*** function below, how should the return type be defined?

  ```
  template <class Func, class T>
  ??? trace(Func f, T t) { std::cout << "Calling f"; return f(t); }
  ```

  Currently, return types that depend on the function argument types are expressed as (complicated) metafunctions that define the mapping from argument types to the return type. For example:

  ```
  template <class Func, class T> typename Func::result_type trace(Func f, T t);
  ```

  or following a recent library proposal [Gre03]:

  ```
  template <class Func, class T>
  typename result_of<Func(T)>::type trace(Func f, T t);
  ```

  Such mappings rely on programming conventions and can give incorrect results. It is not possible to define a set of traits classes/metafunctions that cover all cases. With ***typeof*** (that has appropriate semantics, see Section 3) the ***trace*** function could be defined as:

  ```
  template <class Func, class T>
  auto trace(Func f, T t) -> typeof(f(t));
  ```

  Note the suggested new function definition syntax, discussed in Section 6, where the return type expression following the −> symbol comes after the argument list. Using this syntax, the argument names are in scope in the return type expression.

  As another example, the return types of operators in various algebraic libraries (computations on vectors, matrices, physical units, etc.) commonly depend on the argument types in non-trivial ways. We show an addition operator between two matrices as an example:

  ```
  template <class T> class matrix;
    ...
  template <class T, class U>
  ??? operator+(const matrix<T>& t, const matrix<U>& u);
  ```

For instance, suppose the return type of ***matrix<int>() + matrix<double>()*** is ***matrix<double>***. Expressing such relations requires heavy template machinery. Using ***typeof***, the relation could be expressed as:

```
template <class T, class U>
auto operator+(matrix<T> t, matrix<U> u) -> matrix<typeof(t(0,0)+u(0,0))>;
```

- Often the type of a relatively simple expression can be very complex. It can be tedious to explicitly write such types, making it tedious to declare variables. Common cases are iterator types of containers:

```
template <class T>
int foo(const std::map<T, std::map<T, T>& m) {
  std::map<T, std::map<T, T> >::const_iterator it = m.begin();
  ...
}
```

Types resulting from invocations of function templates can be too complicated to be practical to write by hand. For example, the type of the Lambda Library [JP02] expression ***_1 + _2 + _3*** spans several lines, and contains types that are not part of the public interface of the library. A ***typeof*** operator can be used to address this problem. For example, the declaration of ***it*** using ***typeof*** becomes:

```
typeof(m.begin()) it = m.begin();
```

This is an obvious improvement. However, the semantics of ***typeof*** is not well-suited for the purpose of declaring variables (see Section 5). Furthermore, the redundant repetition of the initializer expression is a distraction and not quite harmless. For example, the following example appeared (innocently) in a reflector discussion:

```
typeof(x*y) z = y*x;
```

The snag is that the types of similar, yet different, expressions are not necessarily the same. Thus, the need to repeat the initializer becomes a maintenance problem. Consequently, we propose a separate mechanism for declaring variables, ***auto***, which deduces the type of the variable from its initializer expression:

```
auto it = m.begin();
```

- The semantics chosen for ***auto*** in variable declarations naturally extends to other conctexs as well. We propose allowing its use in the return type expression of a function stating that the return type should be deduced from the return statement in the body of the function. This can increase readability of code containing short functions, which are frequent in OO programming. For example:

```
template <class T, class U>
auto foo(T t, U u) { return t + u; }
```

3 Design alternatives for *typeof*

Two main options for the semantics of a ***typeof*** operator have been discussed: either to preserve or to drop references in types. For example:

```
int& foo();
...
typeof(foo());   // int& or int?

int a;
int& b = a;

typeof(a);       // int& or int?
typeof(b);       // int& or int?
```

A reference-dropping ***typeof*** always removes top-level references. Some compiler vendors (EDG, Metrowerks, GCC) provide a ***typeof*** operator as an extension with reference-dropping semantics. This appears to be a reasonable semantics for expressing the type of variables (see Section 5). On the other hand, the reference-dropping semantics fails to provide a mechanism for exactly expressing the return types of generic functions, as demonstrated by Stroustrup [Str02]. This implies that a reference-dropping ***typeof*** would cause problems for writers of generic libraries. A reference-preserving ***typeof*** has been proposed to return a reference type if its expression operand is an *lvalue*. Such semantics, however, could easily confuse programmers and lead to surprises. For example, in the above example ***a*** is declared to be of type ***int***, but under a ***typeof*** reflecting "lvalueness", ***typeof(a)*** would be ***int&***. It seems that variants of both semantics are required, thus suggesting the need for two different ***typeof***-like operators. This proposal defines just one ***typeof*** like operator, which attempts to provide the best of both worlds. The discussion in Section 4.1 demonstrates that this is not entirely without problems.

In the standard text (Section 5(6)), 'type of an expression' refers to the non-reference type[1]:

> If an expression initially has the type "reference to ***T***" (8.3.2, 8.5.3), the type is adjusted to ***T*** prior to any further analysis, the expression designates the object or function denoted by the reference, and the expression is an lvalue.

For example:

```
int x;
int xx = x;          // type of the expression x is int
int& y = x;
int yy = y;          // type of the expression y is int
int& foo();
int zz = foo();      // type of the expression foo() is int
```

The lvalueness of an object is expressed separate from its type. In the program text, however, a reference is clearly part of the type of an expression. From here on, we refer to the type in the program text as the *declared type* of an object.

```
int x;               // declared type of x is int
int& y = x;          // declared type of y is int&
int& foo();          // declared type of foo() is int& (because the declared return type of foo is int&)
```

The first line above demonstrates that the lvalueness of an object does not imply that the declared type of the object is a reference type. The semantics of the proposed version of the ***typeof*** operator reflects the declared type of the argument. Therefore, we propose that the operator be named ***decltype***.

4 The *decltype* operator

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()***. However, we do not propose any such extensions. Syntactically, ***decltype(e)*** is treated as if it were a *typedef-name* (cf. 7.1.3). The semantics of the ***decltype*** operator is described as:

1. If ***e*** is a name of a variable in namespace or local scope, a static member variable, or a formal parameter of a function, ***decltype(e)*** is the declared type of that variable or formal parameter. In particular, ***decltype(e)*** results in a reference type if, and only if, the variable or formal parameter is declared as a reference type.

2. If ***e*** refers to a member variable, ***decltype(e)*** is the declared type of the member variable. This rule applies to the following expression forms:

[1] The standard is not always consistent in this respect; in some occasions reference is part of the type.

(a) ***e*** is an identifier that names a member variable and ***e*** is within a definition, i.e., function body, of a member function.

(b) ***e*** is a class member access expression (invocation of the built-in **.** or −> operators) referring to a member variable.

3. If ***e*** is an invocation of a function or of an operator, either user-defined or built-in, ***decltype(e)*** is the declared return type of that function. The standard text does not list the prototypes of all built-in operators. For the functions and operators whose prototypes are not listed, the declared type is a reference type whenever the return type of the operator is specified to be an lvalue, except when rule 2 applies.

4. If ***e*** is an rvalue literal or type ***T***, then ***decltype(e)*** is the non-reference type ***T***. If ***e*** is an lvalue literal of type ***T***, then ***decltype(e)*** is the reference type ***T&***.

5. If ***e*** is an lvalue of function type ***T***, ***decltype(e)*** is ***T&***.

6. ***decltype*** does not evaluate its argument expression.

Note that unlike the ***sizeof*** operator, ***decltype*** does not allow a type as its argument.
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&

decltype(1+2)       // int

int i;
decltype (i = 5)    // int&, because the "declared type" of integer assignment is int&

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*
 ...
}
```

- Function types:

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

- 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 exactly the declared type of the member variable that the expression refers to. Particularly, whether the expression is an lvalue or not does not affect the type. Furthermore, 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.[2]

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

 void foo() {
  decltype(a);   // int
  decltype(b);   // int&
  decltype(c);   // int
 }

 void bar() const {
  decltype(a);   // int (const is not added)
  decltype(b);   // int&
  decltype(c);   // int
 }
 ...
};
```

[2] This decision is based on the reasoning that the ***decltype*** of any expression referring to a member variable gives the type visible in the program text similarly to non-member variables. We have not found particularly strong arguments favoring the proposed semantics over one where cv-qualifiers of the object/pointer expression would affect the declared type of a member variable.

```
A aa;
const A& caa = aa;

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

Note that the **.*** and **->*** operators follow the ***decltype*** rule 3 for functions, instead of rule 2 for member variables. The signatures for these built-in functions are not defined in the standard, hence the lvalue/rvalue rule applies. 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&
```

The operators **.*** and **.** (respectively **->*** and **->**) can thus give different results when querying the type of the same member. Section 4.1 discusses similar cases with other operators and explains why, nevertheless, the proposed rules were chosen. Here we note a useful observation: rule 2 is a special case which only applies for data member accesses where the reference to the member is by the name of the field. The right hand sides of **.*** and **->*** operators are not names of fields but rather *pointer-to-member* objects, and can in fact be arbitrary expressions that result in pointers to members, making it natural to apply rule 3 for functions. Furthermore, **->*** can be freely overloaded, and thus for user-defined ***operator*->*** the function rule must be followed anyway.

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.

- ***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)&       // the redundant & is ok
  const decltype(j)  // the redundant const is ok
  ```

Catering to library authors The semantics of ***decltype*** described above allow to return types of forwarding functions to be accurately expressed in all cases. The ***trace*** and matrix addition examples in Section 2 work as expected with this definition of ***decltype***.

Catering to novice users The rules are consistent; if ***expr*** in ***decltype(expr)*** is a variable, formal parameter, or refers to a member variable, the programmer can trace down the variable's, parameter's, or member variable's declaration, and the result of ***decltype*** is exactly the declared type. If ***expr*** is a function invocation, the programmer can perform manual overload resolution; the result of the ***decltype*** is the return type in the prototype of the best matching function. The prototypes of the built-in operators are defined in the standard, and if some are missing, the rule that an lvalue has a reference type applies. There are some less straightforward cases though, as discussed in the next section.

4.1 Problems with *decltype*

The somewhat subtle properties of ***decltype*** in a few corner cases described in this section were presented to the EWG in Kona. Since the Kona meeting, no changes that would affect the described properties has been made to the proposed rules of ***decltype***. The properties do logically follow from the ***decltype*** rules and it seems that attempts to add more special cases would complicate the rules and implementations unnecessarily. It is also questionable, whether such special cases would lead to more intuitive semantics.

The key property that is required for generic forwarding functions is to have a ***typeof*** mechanism that does not lose information. Particularly, information on whether a function returns a reference type or not must be retained. The following example demonstrates why this is crucial:

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

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

int i; float f;
forward_to_foo(i); // should return int&
forward_to_foo(j); // should return float
```

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(foo(i), foo(f)); // float
forward_foo_to_comma(foo(f), foo(i)); // int&
```

This 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 ***decltype*** operator obeys this rule except for non-member variables and expressions referring to member variables. The deviation from the rule, however, is not serious, as it only occurs with certain syntactic forms and can be accounted for by library solutions (it is possible to emulate the full reference preserving ***typeof*** operator with ***decltype***). The deviation, however, leads to subtle behavior with some built-in operators. Section 2 gave such examples for the $.*$ and $->*$ operators. Comma and conditional operators are subject to the same kinds of subtleties:

```
int i;
decltype(i); // int
```

but

```
decltype(0, i); // int&
decltype(true ? i : i); // int&
```

The ***decltype(i)*** case is covered by ***decltype*** rule 1. Rule 3, however, applies in the latter two cases. In the first of these cases, the topmost expression is an invocation to the built-in comma operator. There is no prototype for that operator, hence the lvalue/rvalue rule applies; since ***i*** is an lvalue, the result is a reference type. The second case follows the same reasoning. In short, the intent of the lvalue/rvalue rule is that if a built-in operator returns an lvalue of some type ***T*** and the standard does not specify its signature, then ***decltype*** acts as if there was a signature for that operator with return type ***T&***.

The member function rules of ***decltype*** can lead to even more surprising cases:

```
struct A {
  int x;
};

const A ca;
decltype(ca.x); // int
decltype(0, ca.x) // const int&
```

The *type*, not the *declared type*, of ***ca.x*** is ***const int*** and ***ca.x*** is an lvalue; thus, ***decltype*** acts as if there was a signature of the comma operator returning a reference type. Hence, the result of ***decltype*** in this case is ***const int&***.

It seems that at least the comma, conditional, $*.$, and $->*$ operators suffer from these subtleties, but potentially surprising cases can arise with other operators as well:

```
int i;
decltype(i); // int
decltype(i = i); // int&
decltype(*&i); // int&
```

It is conceivable that these operators would be handled in some special manner. Such a rule for the comma operator would define ***decltype(a, b)*** as ***decltype(b)*** if the operator invocation resolved to the built-in comma operator. This rule would 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.

Special rules for the conditional operator would be more complex than for the comma operator. In any case, the ***decltype*** rules would get complicated with special cases for comma, conditional, $.*$, and $->*$ operators. Furthermore, there do not seem to be clear criteria that would define this exact set of operators as subject to special rules.

Note that not special casing these operators (particularly comma) gives an easy way to emulate full reference-preserving ***typeof*** semantics, though in a somewhat hackish form. With ***v*** some ***void*** expression, ***decltype(v, e)*** is

equivalent to the full reference-preserving ***typeof*** of ***e***. The ***void*** expression is needed to guarantee that the built-in comma operator is the only matching operator.

It seems that the subtleties described in this section are unavoidable for a ***typeof*** operator that is not either fully reference-preserving, or fully reference-stripping. Hence, the options for ***typeof*** semantics boil down to the following three (banning the use of ***decltype*** with the problematic operations is a fourth option, though not particularly appealing):

1. A reference preserving ***typeof***.
   - Has the right semantics for forwarding functions.
   - Unintuitive results for non-reference variables and member variables.
   - Simple rules.
2. A reference stripping ***typeof***.
   - Intuitive and easy to teach.
   - Simple rules.
   - Useless (almost) for forwarding functions, which is the main motivation for the whole feature.
3. Decltype
   - Intuitive for the most part. Does have some very subtle properties.
   - More complex rules.
   - Adequate for forwarding functions.

In [JS03], we wrote:

> Although this proposal brings the ***decltype*** solution forward, we feel that a careful consideration of the trade-offs between the ***decltype*** solution and the reference-preserving ***typeof*** is necessary.

Regarding this tradeoff, the EWG discussions in Kona and support for ***decltype*** indicated by the straw votes strengthened our belief that the current ***decltype*** specification best hits the 'sweet-spot' of both the needs of advanced generic library authors' and the needs of application programmers'.

5 Auto

Stroustrup brought up the idea of reviving the ***auto*** keyword to 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
```

auto is faced with the same questions as ***typeof***. Should references be preserved or dropped? Should ***auto*** be defined in terms of ***decltype*** (i.e., is ***auto var = expr*** equivalent to ***decltype(expr) var = expr***)? We suggest that the answer to that question be "no" because the semantics would be surprising, non-ideal for the purpose of initializing variables, and incompatible with current uses of ***typeof***. Instead, we propose that the semantics of ***auto*** follow exactly the rules of template argument deduction. The ***auto*** keyword can occur in any deduced context in an 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 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?
```

A unconditionally reference-preserving ***auto*** (e.g. an ***auto*** directly based on ***decltype***) would favor an object-oriented style of use to the detriment of types with value semantics. 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 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. The current template syntax provides such capabilities. It is conceivable that template syntax was allowed for variable declarations. For example, the variable declaration:

```
template <class T> std::pair<T, T> z = bar();
```

would succeed as long as the result type of ***bar()*** would match ***std::pair<T, T>***. We do not propose such a feature at this point.

5.1 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.

5.2 Functions with implicit return types

We suggest the use of ***auto*** to be allowed in a return type expression of a function, or function template, leaving the return type to be deduced from the return statement in the body of the function. For example:

```
template <class T, class U>
auto add(T x, U y) { return x + y; }
```

The return type is deduced as the type deduced for the variable ***ret*** in the expression ***auto ret = x + y***. Any deduced context is allowed:

```
const auto* foo(...) { return expr; }
auto& bar(...) { return expr; }
vector<auto> bah(...) { return expr; }
```

The return types of the above functions are deduced as the types deduced for the variables ***ret1***–***ret3***, respectively:

```
const auto* ret1 = expr;
auto& bar ret2 = expr;
vector<auto> ret3 = expr;
```

Note that the use of ***auto*** as a return type follows exactly the same rules as the use of ***auto*** with variable declarations. Particularly, the return type is not deduced according to the semantics of ***decltype***, which could easily lead to subtle errors. For example:

```
auto foo() {
  int i = 0;
  return ++i;
}
```

With ***decltype*** semantics the return type of the above function would be ***int&***, leading to an attempt to return a reference to a local variable. Hence, ***auto*** as a return is not a tool for forwarding functions, but rather aimed for everyday programming. It provides easier and more convenient means to define short (and often inlined) functions, which are common in OO and generic programming.

We say that functions which have one or more occurrences of ***auto*** in their return type expression have an *implicit return type*. Implicit return types raise some questions:

- Multiple return statements are a problem. The two solutions are either not allowing ***auto*** in the return type of a function with more than one return statement, or applying type deduction rules similar to those used for deducing type of an invocation of the conditional operator. We suggest that functions relying on implicit return types can contain at most one return statement.
- Missing return statement. Should the return type be ***void***, or should such a function definition be an error? We suggest that a function with an implicit return type has the return type ***void*** if the function does not contain a return statement or contains the empty return statement ***return;***
- To be able to deduce the return type from the body of the function, the body needs to be accessible. This restricts a function with an implicit return type to be callable only from the compilation unit that contains the definition of the function.

5.3 Implicit templates

Implicit templates were not strongly supported in EWG discussions in Kona. The main points against were the difficulties in adapting current implementations, and doubts whether the benefits of a new template syntax are worth the costs. We do not suggest this feature for standardization at this point, but nevertheless document the feature here, and describe the corresponding changes and additions to the standard in Section 8.6.

By defining the semantics of ***auto*** in terms of initialization we automatically define ***auto*** in every context where a type is deduced through the initialization rules. Using ***auto*** as a mechanism for *implicit template functions* was suggested in [Str02] and has been discussed within the Evolution Working Group. For example, the implicit template function:

```
void f(auto x) { ... }
```

is equivalent to

```
template<class T> void f(T x) { ... }
```

and

```
void f(auto x, auto y) { ... }
```

is equivalent to

```
template<class T, class U> void f(T x, U y) { ... }
```

The translation from implicit templates to traditional templates is straightforward: every occurrence of ***auto*** is regarded as a new unique template parameter. Note that the set of types that match a particular argument can be constrained in the same ways as with traditional templates. For example,

```
void foo(auto a, auto& b, const auto& c, pair<int, auto> d, auto* e);
```

is equivalent to:

```
template<class A, class B, class C, class D, class E>
void foo(A a, B& b, const C& c, pair<int, D> d, E* e);
```

However, the implicit template syntax cannot capture relations between template arguments. To express such relations, the traditional syntax must be used:

```
template<class T> void f(T x, T y) { ... }
```

6 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.

From the syntaxes proposed in [Str02], and those discussed within the evolution group in the Oxford-03 meeting, the original decltype proposal [JSGS03] suggested adding a new keyword ***fun*** 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> fun add(T t, U u) −> decltype(t + u);
class A {
  fun f() const −> int throw ();
};
```

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

Adding a new keyword is a drastic measure. Therefore, we suggest an alternative syntax that achieves the same goals, but does not necessitate the introduction of a new keyword:

```
auto function−name(parameter−list) −> type
```

A type follows −>, and specifies the return type of the function. Particularly, the type can be expressed using a ***decltype*** expression. For example:

```
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);
```

7 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. The template argument deduction rules form the backbone of two different features: functions with implicit return types, and the use of ***auto*** in variable declarations. Both uses of ***auto*** thus build on the same mechanism and essentially provide new use for what is already in the language.

8 Proposed wording

8.1 `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:

> 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 a name of a variable in namespace or local scope, a static member variable, or a formal parameter of a function, `decltype(e)` is the declared type of that variable or formal parameter. 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.
> 2. If `e` refers to a member variable, `decltype(e)` is the declared type of the member variable. This rule applies to the following expression forms:
>   (a) `e` is an identifier that names a non-static member variable and `e` is within a definition, i.e., function body, of a non-static member function.
>   (b) `e` is a class member access expression (invocation of the built-in `.` or `->` operators) referring to a member variable.
>
>   [*Note:* The l- or rvalueness and the cv-qualification of the *object expression* ([expr.ref]) do not affect the declared types of member variables.]
> 3. If `e` is an invocation of a function or of an operator, either user-defined or built-in, `decltype(e)` is the declared return type of that function or operator. The standard text does not list the prototypes of all built-in operators. For those functions and operators whose prototypes are not listed, except those covered by rule 2, the declared return type is a reference type if, and only if, the return type of the operator is specified to be an lvalue.
> 4. 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&`.
> 5. If `e` is an lvalue of function type `T`, `decltype(e)` is `T&`.
>
> A `decltype` type expression does not evaluate its argument expression. A `decltype` type 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).

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]).

8.2 Removing `auto` as storage specifier: proposed text

Section 3.7.1 Static storage duration [basic.stc.static]

Remove reference to `auto` in paragraph 1, in

> Local objects explicitly declared `auto` or `register` or ...

Section 7.1.1 Storage class specifiers [dcl.stc]

Remove `auto` as a storage class specifier and the discussion about `auto` from paragraph 2.

Section 8.3 Meaning of declarators [dcl.meaning]

From paragraph 2, remove `auto` from the list of type specifiers that apply to *declarator-id*s.

Section 9.2 Class member [class.mem]

Remove `auto` from paragraph 5 which reads:

> A member shall not be auto, extern, or register.

Section A.6 Declarations [gram.dcl.dcl]

Remove `auto` from storage class specifiers.

8.3 `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* in which each *init-declarator* 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, y = 3.14;
> auto u = 5, *v = expr;
> pair<int, auto> a = pair<int, int>(), *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 each 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*]

8.4 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.

8.5 Allowing `auto` to occur in the return type of a function to deduce its return type from its body: proposed text

Section 8.3.5 Functions ([dcl.fct])

At the end of paragraph 1, add the sentence:

> If "*derived-declarator-type-list*" specifies a type scheme (i.e. contains the keyword `auto`), either a function body must follow the declaration ([dcl.fct.def]) or `->` and a return type must follow the declaration, or the program is ill-formed.

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

Add to paragraph 1 that describes the allowed *declarator* forms 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*$_{opt}$
> `auto` *declarator−id* `(` *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. If *decl-specifier-seq* specifies a *type-scheme* (contains occurrences of `auto`) and the declarator is of the first form above, the return type of the function is deduced from the *function-body* according to the following rules:
>
> 1. If the function body contains more than one `return` statement, the program is ill-formed.
> 2. If the sole return statement of the function body is of the form `return e`, where `e` is a non-`void` expression, the return type of the function is the type that would be `deduced` to the *declarator-id* in `D1` in the following declaration.
>
>   *decl−specifier−seq* `D1 = e`
>
> 3. If *decl-specifier-seq* `D1` is of the form `auto` *declarator-id*, the function body is allowed to contain no return statement, an empty return statement `return;`, or a return statement of the form `return e;` where `e` is an expression of type `void`. In these cases, the return type of the function is `void`.

8.6 Allowing `auto` in the types of function parameters: proposed text

Section 8.3.5 Functions ([dcl.fct])

Add to paragraph 3 (before discussion of the transformations to the parameter types):

> If the type of a parameter contains occurrences of `auto`, the function declaration is a *template-declaration* that declares a function template, as described in ([temp]).

Section 14 Templates ([temp])

In paragraph 1, add a new form for *template-declaration*:

> *template−declaration*:
> `...`
> `export`$_{opt}$ *implicit−function−template*

and text:

> *implicit-function-template* ([temp.fct]) is a *declaration* that declares or defines a function whose list of parameter types contains one or more occurrences of `auto`.

Section 14.5.5 Function templates ([temp.fct])

Add a paragraph after paragraph 1:

> A function declaration, or a template declaration declaring a function template, whose list of parameter types contains one or more occurrences of `auto` is an *implicit-function-template*. Each such use of `auto` is an *implicit template parameter*. An implicit function template is treated as if it was defined as a function template obtained with the following procedure.
>
> 1. If the implicit function template does not have a template parameter list, add an empty template parameter list.
> 2. Examine the function parameter list from left to right lexically.
> 3. For each occurrence of `auto` in the parameter list, add a new unique *type* template parameter[3] at the end of the template parameter list, and replace `auto` with the name of the new template parameter.
>
> [*Example:*
>
> ```
> auto& f(auto& i, const pair<auto, auto> j) { ... }
> ```
>
> is equivalent to the definition:
>
> ```
> template <typename __T1, typename __T2, typename __T3>
> auto& f(__T1& i, const pair<__T2, __T3> j) { ... }
> ```
>
> *Note:* the `auto` in the return type expression is not affected.
>
> ```
> template <class T, class U = int, class V>
> int g(T t, auto* x, V v) { ... }
> ```
>
> is equivalent to the definition:
>
> ```
> template <class T, class U = int, class V, class __T1>
> int g(T t, __T1* x, V v) { ... }
> ```
>
>]

References

[Dim01] Peter Dimov. *The Boost Bind Library*. Boost, 2001. `www.boost.org/libs/bind`.

[Gre03] Douglas Gregor. A uniform method for computing function object return types. C++ standards committee document N1437=03-0019, February 2003.

[JP02] Jaakko Järvi and Gary Powell. *The Boost Lambda Library*, 2002. `www.boost.org/libs/lambda`.

[JPL03] Jaakko Järvi, Gary Powell, and Andrew Lumsdaine. The Lambda Library: unnamed functions in C++. *Software—Practice and Experience*, 33:259–291, 2003.

[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`.

[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.

[3] `auto` cannot be used in place of non-type template parameters

[Vel] Todd Veldhuizen. Blitz++ home page. `http://oonumerics.org/blitz`.

[WK02] Jörg Walter and Mathias Koch. *The Boost uBLAS Library*. Boost, 2002. `www.boost.org/libs/numeric`.

9 Acknowledgments

We are grateful to Jeremy Siek, Douglas Gregor, Jeremiah Willcock, Gary Powell, Mat Marcus, Daveed Vandevoorde, Gabriel Dos Reis, 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.