Mechanisms for querying types of expressions: Decltype and auto revisited

Programming Language C++

Document no: N1527=03-0110

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

September 21, 2003

1 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. A subsequent proposal [JSGS03] 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 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. This proposal builds on the two earlier proposals taking into account discussions in the C++ standard reflector. The major differences with [JSGS03] are:

- Semantics for ***decltype*** with member variables has changed.
- Introducing a new keyword (***fun***) is not necessary.
- The applicability of ***auto*** has been expanded considerably. We explore the uses of ***auto*** for implicit templates and functions where the return type is deduced from the body of the function. To take the full advantage of the proposed features, we suggest new syntax for function declarations.
- Some subtleties of the proposed ***decltype*** semantics, which arise with certain built-in operators, are described.

This proposal attempts to generalize the ***auto*** related features as much as possible, even though it may be desirable to adopt the features in a more limited form. Working out the most general cases in detail is beneficial for better understanding the design space of the features and the implications of the proposal.

To make the proposal self-contained we include background material from [JSGS03], 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***.

1.1 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 2) 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 5, 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 4). 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 C++ template syntax is verbose, and a topic of continued criticism. Implicit templates and automatic deduction of the return type based on the body of the function would allow a much more concise syntax for most function templates. This would significantly increase the readability of code containing short functions, which are frequent in OO programming. Using the features described in this proposal, the ***pair_incr*** function:

```
template <class T, class U>
inline pair<T, U> pair_incr(pair<T, U>& p) { ++p.first; ++p.second; return p; }
```

can be written as:

```
inline auto pair_incr(pair<auto, auto>& p) { ++p.first; ++p.second; return p; }
```

2 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 4). 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 3.1 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.

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

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***.

3 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. Particularly, ***decltype(e)*** results in a reference type only 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:

   (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 a literal, ***decltype(e)*** is a non-reference type.

5. ***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)
```

  Note that objects of function types cannot exist:

```
decltype(foo) f1 = foo;      // error, we can't have a variable of type int(char)
decltype(foo)* f2 = foo;    // fine: f2 is an int(*)(char)
decltype(foo)& f3 = foo;   // fine: f3 is an 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
 }
 ...
};

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&
```

[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.

The operators **.*** and **.** (respectively **->*** and **->**) can thus give different results when querying the type of the same member. Section 3.1 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:

  The lvalueness or rvalueness of a literal has no bearing on the result of ***decltype*** applied to a literal. The declared types of all literals are non-reference types.

```
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 1 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.

3.1 Problems with *decltype*

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.

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. Particularly, it is not clear whether ***auto*** (see Section 4) could be the tool for everyday programming leaving any ***typeof*** operator as an advanced feature for authors of generic libraries. Hence, an analysis of use cases for a ***typeof*** operator (other than forwarding functions) is needed.

4 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. Therefore we suggest that allowing template specializations for variable declarations be considered. For example, the variable declaration:

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

would succeed as long as the result type of pair would match ***std::pair<T, T>***. Hence,

```
std::pair<auto, auto> y = bar();
```

would be equivalent to

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

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.

4.2 Implicit templates

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) { ... }
```

Hence, the longer (often criticized) heavyweight template notation will only be needed in these cases, or if one needs to name a template parameter

4.3 Functions with implicit return types

Going further into the same direction, ***auto*** can be used as the return type of a function:

```
auto add(auto x, auto 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, and as implicit template parameters. 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 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. Let ***auto−type*** be a type expression containing one or more occurrences of ***auto***, such as ***auto***, ***const auto&***, or ***pair<auto, auto>***. We suggest the following function declaration syntaxes to be allowed (names of the syntactic parts are not from the standard):

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

The first syntactic form gives the function the type that would be deduced for the variable ***ret*** in the expression:

```
auto−type ret = expression;
```

Examples:

```
auto f(int i) -> g(i) { return g(i); }
auto& id(auto& a) -> a { return a; }
```

The exact same set of rules applies as for variable declarations and implicit templates.

In the second syntactic form, 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;
auto id(auto& a) -> decltype(a);
```

Note that in this syntactic form, only the keyword ***auto*** is allowed before the function name, instead of allowing any ***auto−type***.

We propose the following set of rules for the new kind of function declarations:

- A function declaration (that is not followed by a function body) must specify the return type explicitly using any function declaration syntax. Particularly, omitting the return type is an error rather than defaulting to ***void***.
- 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.

The following function declarations and definitions give examples of the application of the above rules:

```
void bar(auto a);

auto foo(auto a) -> bar(a); // ok, void

auto foo(auto a) { return bar(a); } // error, missing return type
                                    // and previous declaration specified one

auto foo(auto a) -> decltype(bar(a)); // ok, void

void foo(auto a); // ok, void
```

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

7 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.

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.

[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.

[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`.