N2337=07-0197
2007-08-01
Daveed Vandevoorde
daveed@edg.com

# The Syntax of auto Declarations

## 1 The Issues

Two proposed new uses of the keyword **auto** (one of which is already in the Working Paper) are raising new parsing ambiguity issues. The issues arise primarily because in a declaration an identifier followed by a parenthesis can mean one of three things (not considering cases where the identifier is actually part of an expression within the declaration):

- a type name followed by a nested declarator;
  e.g., **X(*p)[2]**
- a variable name followed by a parenthesized initializer;
  e.g., **Complex X(1, 2)**
- a function name followed by a parenthesized list of parameter declarations;
  e.g., **void X(int, int)**

Let's first examine in some detail the ambiguities raised by the two new uses of **auto**.

### 1.1 Known Problem: The auto type specifier

This issue is also Core Issue 629.

Consider the following bit of code:

```
typedef int T;
int x1, x2, y;
void f1() {
  auto T(&x1) = y;  // (1)
  auto T(&x2);      // (2)
}
```

In C++98, declaration (1) is a valid declaration of a reference **x1** to an object of type **T** (the object is **y** in this case), whereas (2) is an error because it attempts to declare a reference **x2** without any initialization.

In C++0x, **auto** has a new meaning associated with the following syntactic disambiguation rule (7.1.5.4):

> [1] The **auto** *type-specifier* has two meanings depending on the context of its use. In a *decl-specifier-seq* that contains at least one *type-specifier* (in addition to **auto**) that is not a *cv-qualifier*, the **auto** *type-specifier* specifies that the object named in the declaration has automatic storage duration. The *decl-specifier-seq* shall contain no *storage-class-specifiers*. This use of the **auto** specifier shall only be applied to names of objects declared in a block (6.3) or to function parameters (8.4).

> [2] Otherwise (**auto** appearing with no type specifiers other than *cv-qualifiers*), the **auto** *type-specifier* signifies that the type of an object being declared shall be deduced from its initializer. The name of the object being declared shall not appear in the initializer expression.

The current working paper includes words that make the **T** in our example a type specifier (7.1):

> [2] The longest sequence of *decl-specifiers* that could possibly be a type name is taken as the *decl-specifier-seq* of a *declaration*. […]

These words arguably need work (is **static inline int** a "type name"?) but the intent seems unambiguous: **T** in our example is part of the *decl-specifier*s and not part of the declarator. That in turn implies that the meaning of the example is unchanged from C++98. Unfortunately, that outcome is surprising in the context of C++0x where **auto** is likely to be used exclusively for type deduction purposes (paragraph [2] in 7.1.5.4). Other examples may seem even more surprising:

```
typedef int T;
void f2() {
  auto T = 3;  // (3) Error.
}
```

On the other hand, making all cases like (2) and (3) be valid with an intuitive meaning either breaks backward compatibility with C++98, or requires heroic parsing strategies.

## 1.2 Potential problem: New function declarator syntax

N1978 "Decltype (revision 5)" proposes a new function declaration syntax that moves the return type from the decl-specifier sequence to the declarator. The proposal suggests using the keyword auto as a placeholder "type specifier" in such cases. For example[1]:

```
// Function f() returning int:
auto f() -> int;
// Function type F returning pointer to int:
typedef auto F() -> int*;
// Pointer to function pf, that:
//  - takes a pointer to a function returning an int,
//  - return a pointer to a function returning an int.
auto *pf(auto *p() -> int) -> auto *()->int;
```

Consider now the following case illustrating what looks like an ambiguity:

```
typdef int T;
auto f(T()) -> int;
```

Until the **->** token is seen, this looks like a variable declaration with a deduced type. However, this is a case where the declaration-vs.-expression disambiguation rules apply, and hence the **T()** is not treated as an expression, but as a declaration of a parameter of pointer-to-function type.

---

[1] The details of the syntax are not yet fully specified and N1978 no longer reflects the latest thinking on the matter. So the example may not end up being valid C++0x.

Although the details of the new function declarator syntax are not yet known, it is nonetheless clear that reusing the **auto** specifier for that purpose constrains the design choices and/or increases the potential to create additional ambiguities. Furthermore, those ambiguities may be the interpretation of **auto** as a deducible type specifier rather than auto as a storage class indication.

# 2 Possible Resolutions

The following subsections describe a few alternatives to address the known problem.

## 2.1 Drop auto for storage class specification

The use of **auto** to denote the storage class of a local variable (the only valid C++98 use) is superfluous in that a C++98 auto declaration will not change meaning if the **auto** keyword is simply dropped. Furthermore, searching for uses of **auto** using tools like Google Code Search (http://google.com/codesearch) suggest that in C++ **auto** is used very little (Google Code Search finds less than 50 uses of **auto** in C++ code).

This suggests that a simple solution to the ambiguities between **auto** as a storage class and **auto** as a deducible type is to drop the C++98 meaning of **auto** altogether, and only leave the new meaning as a deducible type specifier. This may be somewhat controversial, in part because a language feature would be dropped without going through a "deprecated stage". On the other hand, it would greatly simplify the programming model. Subsection 2.4 below also demonstrates that some valid C++98 cases can change meaning with this approach.

A potential shortcoming of this option is that it may not resolve ambiguities that might arise with the use of **auto** to introduce a new-style function declaration.

## 2.2 Clarify the status quo

If the known problem presented here is considered minor, we could opt not to address it. Instead, it might be worthwhile to clarify the status quo (e.g., by tightening up the words in 7.1[2]).

The resulting semantics are what is currently implemented in the EDG front end (versions 3.9 and later). It is to the best of our knowledge the only existing implementation experience. It also maintains full backward compatibility with C++98.

One argument against the status quo, however, is that an unrelated declaration in namespace scope (brought in through header-inclusion, for example) may invalidate a local declaration. That is a significant wart on the programming model, but similar situations abound elsewhere in C++ and in this case the problem is easily corrected when it arises. Such problems can also be avoided using extra parentheses (generally regarded as an unattractive style option) or using appropriate naming conventions (although the naming conventions of third-party header files are usually outside the programmer's control).

Again, this approach does not help the potential problem of ambiguities that might arise with the use of **auto** to introduce a new-style function declaration.

### 2.3 Drop parenthesized initializers for auto deduction

A potential solution is to require that parsers pick whichever interpretation of **auto** actually works. If both interpretations work (likely a much rarer situation), either interpretation could be mandated, or the construct could be deemed invalid.

Unfortunately, the added complexity this implies for implementations is unreasonable.

It turns out, however, that the most objectionable situations involve syntax where parentheses could be interpreted as enclosing an initializer expression (as opposed to enclosing a nested declarator). Line (2) in our original example is such a situation. If only initializers introduced with a **=** token can be used with deduced **auto** type specifiers in variable declarations[2], then the cost of looking ahead to determine which interpretation is applicable becomes much more reasonable. In fact, a single-token look-ahead provides the answer: If an identifier is followed by **=** and **auto** has been seen, that identifier is a declarator-id.

Disallowing parenthesized initializers may seem a drastic limitation since direct-initialization is often preferred (and sometimes required) when initializing objects of class type. However, the new brace-based initializer syntax proposed in N2215 ("Initializer lists", B.Stroustrup and D. Dos Reis) provides an alternative to express the same semantics using the **=** token. This option would also preserve full backward compatibility with C++98.

An added attraction of this option is that it almost certainly avoids the potential problem of ambiguities that might arise with the use of **auto** to introduce a new-style function declaration.

### 2.4 Rejected options

The previous section already mentioned the idea that the standard could require "full disambiguation"; i.e., an implementation would effectively try the two interpretations of **auto** and retain the one that results in a valid declaration (if indeed there is only one). The cost of doing so is considered prohibitive, however. It is possible for a construct to be valid under both interpretations:

```
typedef int T;
int n = 3;
void f() {
  auto T(n);
  n = 7;  // Which n is this?
}
```

A disambiguation rule that prefers **auto** to be a deducible type over **auto** being a storage class indication would result in a silent change of meaning. That is also true if the storage class interpretation is dropped altoghether.

Another option is to retain only the classical meaning of **auto** (i.e., as a storage class indication) for a few cases where it is "obviously" the only possible interpretation. This would include *decl-specifier* sequences containing a *simple-type-specifier* that is a

---

[2] The use of **auto** in new-expressions is unambiguous and therefore need not be affected by this rule.

keyword (like **`short`** or **`float`**), or situations where a problematic identifier is followed by another identifier (e.g., in **`auto X Y`** the identifier **`X`** cannot be a valid start of a declarator, and can therefore be assumed to name a type). The Core WG considered such approaches, but concluded they were too "ad hoc".

## 3 Conclusion

Without a clearer picture of the new function declaration syntax, it is difficult to make a recommendation for the best way forward. The following therefore assumes that the proposal for the new function declaration syntax does not introduce new ambiguities.

Despite the unfortunate precedent of dropping a language feature without a transition period where that feature has been deprecated, I recommend dropping the classic use of the **`auto`** specifier altogether. This provides a complete and simple solution for the new meaning of **`auto`**, and I do not expect it to have a significant impact in terms of backward compatibility.

If no consensus can be found for that option or if the new function declaration syntax introduces new ambiguities, my "second preferred" option is to disallow parenthesized variable initializers for variables declared with a deducible **`auto`** type specifier. That option was the consensus resolution of core issue 629 by the Core WG in April 2007 (Oxford). This would require the availability of the newly-proposed brace-based initialization syntax to achieve direct-initialization.