A Proposal to Restore Multi-declarator `auto` Declarations

Document #:	WG21/N1737 = J16/04-0177
Date:	November 2, 2004
Revises:	None
Project:	Programming Language C++
Reference:	ISO/IEC IS 14882:2003(E)

Reply to: Walter E. Brown<`wb@fnal.gov`>
CEPA Dept., Computing Division
Fermi National Accelerator Laboratory
Batavia, IL 60510-0500

Contents

Never make the same mistake once.
— Lois Goldthwaite

1 Introduction

Proposals to introduce a new form of `auto` declaration have received considerable, largely favorable, attention from the C++ standards bodies. This language extension was first described, in draft form, in [Str02], and was successively refined in [JSGS03, JS03, JS04]. However, the proposal's most recent version [JSDR04, presented at the October 2004 meeting in Redmond, Washington, USA] explicitly omits one aspect found in earlier drafts: the ability to define multiple variables (technically, to provide multiple *declarators*) in a single `auto` declaration:

> Rules for the use of `auto` to declare variables have changed to allow only one variable declaration in a declarator list when the type of the variable is deduced from its initializer expression.

Although it was likely well-intentioned, we believe strongly that this omission should be reversed. In this paper, we will reconsider the underlying issues and propose consistent semantics directly addressing the concerns that could not previously be resolved and that led to the feature's consequent abandonment. Along the way, we will shed some new light on a minor (known) inconsistency in the current language, and show that our proposed formulation addresses this inconsistency in the general case as well as in the particular case of an `auto` declaration.

In light of the semantics offered herein, we ask that the excised capability be subsequently restored either by incorporation into all future drafts of the `auto` proposal or by direct vote into the Working Draft [Bec04, as amended] concurrently with the `auto` proposal's final form.

2 Rationale

Among the initial motivation for proposing `auto` was an example intended primarily to demonstrate the convenience aspect of the proposed new use for the `auto` keyword. That example exhibited looping code such as:

```
//                                Listing 1
vector<MyType> v;
// fill v ...
for( auto it = v.begin(); it != v.end(); ++it )  {
  // use *it ...
}
```

In such a context, the simple `auto` would replace today's rather unwieldy equivalent:

```
//                                Listing 2
vector<MyType> v;
// fill v ...
typedef  vector<MyType>::iterator  iter;
for( iter it = v.begin(); it != v.end(); ++it )  {
  // use *it ...
}
```

Now, it is today possible to write such code as:

```
//                                Listing 3
vector<MyType> v;
// fill v ...
typedef  vector<MyType>::const_iterator  iter;
for( iter it = v.begin(), e = v.end(); it != e; ++it )  {
  // use *it ...
}
```

Because the syntax of a `for` statement permits only a single statement in its initialization clause, the decision to disallow multiple variables in an `auto` declaration permits no analog using `auto`'s more convenient notation:

```
//                                Listing 4
vector<MyType> v;
// fill v ...
for( auto it = v.begin(), e = v.end(); it != e; ++it )  {  // not allowed
  // use *it ...
}
```

We believe this is an important use case, and therefore propose to restore the above syntax.

In our view, it seems likely that the recent decision to excise `auto`'s multi-variable capability resulted from an inability to reach consensus on either of the two equally plausible semantics described at the recent Redmond meeting. We will shortly explore (and later reconcile) both semantics, but for purposes of comparison and analysis we will first describe the properties of today's multi-variable declaration semantics,

3 Properties of today's multi-declarator semantics

To provide a basis for comparison, we observe that current declaration syntax for variables of type T takes the general form:

```
//                                Listing 5
T  a, b, c;
```

As [ISO03, footnote 85] points out, the associated semantics are usually applied as-if expanded and thus declared as:

```
//                                Listing 6
T  a;
T  b;
T  c;
```

[ISO03, footnote 85] further notes that the exception to these semantics arises only when a variable given the same name as the name of its underlying type (*e.g.*, a variable named T of type T) is among the variables being declared, and is not in last position. In such circumstances, scope rules interfere with correct interpretation of the otherwise-equivalent expansion shown above.

The above, traditional, interpretation of a canonical multi-variable declaration is characterized by *consistent form* as well as *consistent behavior*. By *consistent form* we mean that each of the expanded declarations shares a common *decl-specifier* (T in our example). *Consistent behavior* follows, in that each variable thus obtains the identical attributes (type, cv-qualification, *etc.*) as induced by that common *decl-specifier*.

In considering the semantics of multi-variable auto declarations of the form:

```
//                                Listing 7
auto  a = ..., b = ..., c = ...;
```

it has to date appeared that invoking type deduction permits us to achieve either consistent form or consistent behavior, but not both in this context. However, we believe that we can now reconcile these two semantics. Before doing so, we will briefly present and explore both of them, as we believe the inability to choose between their desirable characteristics was, in significant part, responsible for the decision to excise multi-variable auto declarations.

4 Consistency in multi-variable auto declarations

Let us now consider the declaration:

```
//                                Listing 8
auto  a = 1, b = 3.14, * c = new float;
```

To achieve consistent form would require that the type of each *declarator* in a multi-variable auto declaration be independently deduced. Alas, this decision leads directly to inconsistency in behavior:

```
//                                 Listing 9
auto   a = 1;          // deduce int; decltype(a) is int
auto   b = 3.14;       // deduce double; decltype(b) is double
auto * c = new float;  // deduce float; decltype(c) is float *
```

As this example shows, preserving consistent form (as-if each *declarator* were independently declared alike, *i.e.*, via auto) produces behavior that is subject to variation on a per-*declarator* basis.

To achieve consistent behavior would require that only the first variable in a multi-variable auto declaration be subject to type deduction, and that the deduced attribute be applied to each remaining *declarator*. However, this decision quickly produces inconsistency in form:

```
//                                 Listing 10
auto          a = 1;          // deduce int; decltype(a) is int
decltype(a)   b = 3.14;       // apply deduced int; decltype(b) is int
decltype(a) * c = new float;  // initialization error (type mismatch)
```

As this example shows, while the resulting deduced attributes (here, type int) are consistently applied, the forms of the expanded declarations differ: the first *declarator* is expanded via auto (and thus subject to type deduction), while the second and any subsequent *declarator*s are expanded via decltype.

5 Discussion

Two observations regarding these behaviors emerged at the 2004 Redmond meeting.

First, it was noted that application of consistent-form semantics would permit a single declaration to produce iterators for a parallel traversal of two distinct types of containers. While we agree with this observation, we disagree as to its significance: Since this is not a capability available today in the absence of the proposed auto, we believe there is no reason to require that it suddenly become available via the new auto.

Second, it was also observed that the consistent-behavior formulation provides a capability that the consistent-form formulation can't express: the ability to declare one or more default-initialized variables of (or related to) the common type:

```
//                                 Listing 11
auto  a = 1, b, * c;  // int a = 1, b, * c;
```

While we are uncertain as to the value of such a capability, we would see no reason to forbid it.

6 Reconciliation, generalization, and proposal

We believe it is possible to achieve both consistent form and consistent behavior. We do so by inserting (for purposes of exposition) an intermediate __Deduced_type definition, and applying this type consistently in the as-if expansion:

```
//                                 Listing 12
typedef  int  __Deduced_type;    // exposition only
__Deduced_type   a = 1;          // decltype(a) is int
__Deduced_type   b = 3.14;       // decltype(b) is int
__Deduced_type * c = new float;  // error; decltype(c) would be int *
```

Not only do we achieve consistency of both form and behavior via such a reconciled formulation, we also address more complicated situations. For example, when the leading *declarator* includes a *ptr-operator*:

```
//                                Listing 13
auto * a = new int(1), b = 3.14, * c = new float;
```

our formulation attaches semantics as-if declared:

```
//                                Listing 14
typedef  int  __Deduced_type;     // exposition only
__Deduced_type * a = new int(1); // decltype(a) is int *
__Deduced_type   b = 3.14;       // decltype(b) is int
__Deduced_type * c = new float;  // error; decltype(c) would be int *
```

Finally, we note that generally describing multi-variable declarations in this manner also captures today's semantics in all cases, without need for any special formulation to handle the (uncommon) case of a *declarator-id* that matches the type name in its *decl-specifier-seq*. Thus, we can optionally reformulate [ISO03, footnote 85] to as to read, for example:

> A declaration with several declarators is always equivalent to the corresponding sequence of declarations, each with a single declarator, following a `typedef` of the common *decl-specifier-seq*. That is,
>
> `T D1, D2, ... Dn;`
>
> is always equivalent to
>
> `typedef T __DSQ; __DSQ D1; __DSQ D2; __DSQ; ... __DSQ Dn;`
>
> in which `__DSQ` stands for `__Decl_Specifier_Sequence` and `T`, if it involves `auto`, is replaced by the type deduced from `D1`'s required initializer as described elsewhere herein.

Because this latest formulation appears to reconcile consistent-form and consistent-behavior semantics, we propose it as the designated semantics of multi-variable `auto` declarations (and, perhaps, of all multi-variable declarations). If this proposal is adopted by the standardization bodies, we will be pleased to assist in drafting proposed wording for incorporation into the Working Draft.

7 Summary and conclusion

In this paper, we have argued in favor of restoring the multi-variable form of `auto` declarations that were dropped from [JSDR04]. By analogy with today's multi-variable declaration syntax, we first described two possible semantics for this feature, one achieving consistent form and the other achieving consistent behavior.

We postulated that this dichotomy, and the attendant difficulty in selecting one over the other, was ultimately responsible for the recent decision to excise the feature entirely. To remedy such omission, we set forth a third formulation of the desired semantics, arguing that this formulation obtains both consistent form and consistent behavior for multi-variable declarations in all cases, including those of `auto` declarations without special treatment. We then proposed this third formulation for incorporation into future revisions of the underlying `auto` proposal, and thence ultimately into the Working Draft.

We respectfully urge the C++ standards bodies to consider this proposal in a time frame consistent with that of the forthcoming C++0X standard.

8 Acknowledgments

Many thanks to my Fermilab colleagues for their able assistance and advice during the drafting of this paper. I wish also to thank the Fermi National Accelerator Laboratory's Computing Division, sponsor of our participation in the C++ standards effort, for its support.

Bibliography

[Bec04] Pete Becker. Unofficial working draft, standard for programming language C++. Paper N1655, JTC1-SC22/WG21, July 16 2004. Online: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1655.pdf`; same as ANSI NCITS/J16 04-0095.

[ISO98] *Programming Languages — C++, International Standard ISO/IEC 14882:1998(E)*. International Organization for Standardization, Geneva, Switzerland, 1998. 732 pp. Known informally as C++98.

[ISO03] *Programming Languages — C++, International Standard ISO/IEC 14882:2003(E)*. International Organization for Standardization, Geneva, Switzerland, 2003. 757 pp. Known informally as C++03; a revision of [ISO98].

[JS03] Jaako Järvi and Bjarne Stroustrup. Mechanisms for querying types of expressions: Decltype and auto revisited. Paper N1527, JTC1-SC22/WG21, September 21 2003. Online: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1527.pdf`; same as ANSI NCITS/J16 03-0110.

[JS04] Jaako Järvi and Bjarne Stroustrup. Decltype and auto (revision 3). Paper N1607, JTC1-SC22/WG21, February 17 2004. Online: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1607.pdf`; same as ANSI NCITS/J16 04-0047.

[JSDR04] Jaako Järvi, Bjarne Stroustrup, and Gabriel Dos Reis. Decltype and auto (revision 4). Paper N1705, JTC1-SC22/WG21, September 12 2004. Online: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1705.pdf`; same as ANSI NCITS/J16 04-0145.

[JSGS03] Jaako Järvi, Bjarne Stroustrup, Douglas Gregor, and Jeremy Siek. Decltype and auto. Paper N1478, JTC1-SC22/WG21, April 28 2003. Online: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1478.pdf`; same as ANSI NCITS/J16 03-0061.

[Str02] Bjarne Stroustrup. auto/typeof. C++ extensions reflector message c++std-ext-5364, October 15 2002.