Deducing the type of variable from its initializer expression (revision 4)

Programming Language C++
Document no: N1984=06-0054

Jaakko Järvi
Texas A&M University
College Station, TX
jarvi@cs.tamu.edu

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

Gabriel Dos Reis
Texas A&M University
College Station, TX
gdr@cs.tamu.edu

2006-04-06

1 Introduction

This document is a revision of the documents N1794=05-0054 and N1721=04-0161. The document N1721=04-0161 contained the suggested wording for new uses of keyword **auto**, which were unanimously approved by the evolution group meeting in Redmond, October 2004. Based on the discussions and straw-polls in the Lillehammer meeting in April 2005, wording for allowing the initialization (with **auto**) of more than one variables in a single statement was added; N1721=04-0161 allowed only one variable initialization per statement. The current document revises the wording of N1794=05-0054 based on technical comments from the core working group from the Lillehammer meeting and repeated reviews in the Mont-Tremblant meeting. Essentially only the suggested wording has changed from N1794=05-0054.

2 Proposed features

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

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

The **auto** keyword can occur as a simple type specifier (allow to be used with cv-qualifiers, `*`, and `&`) and the semantics of **auto** should follow exactly the rules of template argument deduction. Examples (the notation `x : T` in the comments is read as "`x` has type `T`"):

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

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

```
auto& y3 = bar();            // y3 : float&

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

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

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

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

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

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

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

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

Thus, as in the rest of the language, value semantics is the default, and reference semantics is provided through consistent use of `&`.

Multi-variable declarations

More than one variable can be declared in a single statement:

```
int i;
auto a = 1, *b = &i;
```

In the case of two or more variables, both deductions must lead to the same type. Note that the declared variables can get different types, as is the case in the above example. The declarations are handled from left to right. The following code is thus valid:

```
auto x = 1, *y = &x;
```

Obviously, the variable being declared cannot be used in its own initializer. For example, the following declaration is erroneous:

```
auto j = &j;
```

Direct initialization syntax

Direct initialization syntax is allowed and is equivalent to copy initialization, for the purpose of type deduction. 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 an occurrence of of **`auto`**, `v` as a variable name, and `x` an expression, and type deduction proceeds as in the corresponding copy initialization expression `T v = x`. Examples:

```
const auto& y(x) -> const auto& y = x;
```

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.

3 Proposed wording

Section 7.1.1 Storage class specifiers [dcl.stc]

Paragraph 1 should start:

> The storage class specifiers are
>
> *storage−class−specifier :*
> ~~`auto`~~
> `register`
> `static`
> `extern`
> `mutable`

Paragraph 2 should be:

> The ~~`auto` and~~ `register` specifier~~s~~ shall be applied only to names of objects declared in a block (6.3) or to function parameters (8.4). ~~They specify~~ **It specifies** that the named object has automatic storage duration (3.7.2). An object declared without a *storage-class-specifier* at block scope or declared as a function parameter has automatic storage duration by default. ~~[*Note:* hence, the `auto` specifier is almost always redundant and not often used; one use of `auto` is to distinguish a *declaration-statement* from an *expression-statement* (6.8) explicitly. —*end note*]~~

Paragraph 3 should be:

> A `register` specifier ~~has the same semantics as an `auto` specifier together with~~ **is** a hint to the implementation that the object so declared will be heavily used. [*Note:* the hint can be ignored and in most implementations it will be ignored if the address of the object is taken. —*end note*]

Section 7.1.5 Type specifiers [dcl.type]

Paragraph 2 should read:

As a general rule, at most one *type-specifier* is allowed in the complete *decl-specifier-seq* of a *declaration*. The only exceptions to this rule are the following:

- `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) or template type arguments (14.3), in which case the redundant cv-qualifiers are ignored.
- `signed` or `unsigned` can be combined with `char`, `long`, `short`, or `int`.
- `short` or `long` can be combined with `int`.
- `long` can be combined with `double`.
- **`auto` can be combined with any other type specifier, except with itself.**

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 to be deduced

Change paragraph 2 to read:

> **The `auto` specifier is a placeholder for a type to be deduced ([dcl.spec.auto] 7.1.5.4).** The **other** *simple-type-specifier*s specify either a previously-declared user-defined type or one of the fundamental types (3.9.1). Table 7 summarizes the valid combinations of *simple-type-specifier*s and the types they specify.

New Section 7.1.5.4 `auto` specifier [dcl.spec.auto]

This would be a new section, even though `auto` is a simple type specifier.
Paragraph 1 should be:

> **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-specifier*s. 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).**

Paragraph 2 should be:

> **Otherwise (`auto` appearing with no type specifiers other than *cv-qualifiers*), the `auto` *type-specifier* signifies that the type of an object being declared is to be deduced from its initializer. The name of the object being declared shall not appear in its initializer expression.**
>
> **This use of `auto` is allowed when declaring objects in a block [stmt.block] (6.3), in namespace scope [basic.scope.namespace] (3.3.5), or in a *for-init-statement* [stmt.for] (6.5.3). The *decl-specifier-seq* shall be followed by one or more *init-declarator*s, each of which shall have a non-empty *initializer* of either of the following two forms:**

= ***assignment−expression***
(***assignment−expression*** **)**

[*Example:*

```
auto x = 5;                   // ok, x has type int
const auto *v = &x, u = 6;    // ok, v has type const int*, u has type const int
static auto y = 0.0;          // ok, y has type double
static auto int z;            // ill−formed, auto and static conflict
auto int r;                   // ok, r has type int
```

— *end example*]

Paragraph 3 should be:

The `auto` *type-specifier* can also be used in declaring an object in the *condition* of a selection statement [stmt.select] (6.4) or of an iteration statement [stmt.iter] (6.5), in the *type-specifier-seq* in *new-type-id* [expr.new] (5.3.4), and in declaring a static data member with a *constant-initializer* that appears within the *member-specification* of a class definition [class.static.data] (9.4.2).

Paragraph 4 should be:

A program that uses `auto` in a context not explicitly allowed in this section is ill-formed.

Paragraph 5 should be:

Once the type of a *declarator-id* has been determined according to [dcl.meaning], the type of the declared variable using the *declarator-id* is determined from the type of its initializer using the rules for template argument deduction. Let `T` be the type that has been determined for a variable identifier `d`. Obtain `P` from `T` by replacing the occurrence of `auto` with a new invented type template parameter `U`. Let `A` be the type of the initializer expression for `d`. The type deduced for the variable `d` is then the deduced type determined using the rules of template argument deduction from a function call ([temp.deduct.call]), where `P` is a function template parameter type and `A` the corresponding argument type. If the deduction fails, the declaration is ill-formed.

If the list of declarators contains more than one declarator, the type of each declared variable is determined as described above. If the type deduced for the template parameter `U` is not the same in each deduction, the program is ill-formed.

[*Example:*

```
const auto &i = expr;
```

The type of `i` is the deduced type of the parameter `u` in the call `f(expr)` of the following invented function template:

```
template <class U> void f(const U& u);
```

— *end example*]

Section 8.3.4 arrays [dcl.ptr]

Paragraph 1 should start:

> In a declaration `T D` where `D` has the form
>
> `D1 [`*constant−expression$_{opt}$*`]`
>
> and the type of the identifier in the declaration `T D1` is "*derived-declarator-type-list* `T`," then the type of the identifier of `D` is an array type**; if the type of the identifier of `D` contains the `auto` type deduction *type-specifier*, the program is ill-formed.** `T` is called the array *element type*; ...

Currently, the change is thus to ban the use of auto with arrays. This is due to arrays decaying to pointers automatically. For example:

```
int x[5];
auto y[5] = x;
```

Here, expression `x` would decay to a pointer, and would not match the type "auto y[5]". Note that depending on the work on initializers we may wish to revisit this part. For example, we may wish to enable

```
auto x[] = {a, b, c};
```

Also, we can debate whether the following should be allowed:

```
int x[5];
auto y[] = x; // would this be allowed and y : int * ?
```

Section 5.3.4 New [expr.new]

Add the following text after the paragraph 1

> **If the `auto` *type-specifier* appears in the *type-specifier-seq* of a *new-type-id* or *type-id* of a *new-expression*, the *type-specifier-seq* shall contain no other *type-specifiers* except *cv-qualifiers*, and the *new-expression* shall contain a *new-initializer* of the form `(`*assignment-expression*`)`.**
>
> **The allocated type is deduced from the *new-initializer* as follows: Let `(e)` be the *new-initializer* and `T` be the *new-type-id* or *type-id* of the *new expression*, then the allocated type is the type deduced for the variable `x` in the invented declaration ([dcl.spec.auto]):**
>
> ```
> T x = e;
> ```
>
> **[*Example:***
>
> ```
> new auto(1);                  // allocated type is int
> auto x = new auto('a');       // allocated type is char, x is of type char*
> ```
>
> ***— end example*]**

6.4. Selection statements [stmt.select]

Paragraph 2 should be:

> The rules for conditions apply both to *selection-statement*s and to the `for` and `while` statements (6.5). The *declarator* shall not specify a function or an array. The *type-specifier-seq* shall not contain `typedef` and shall not declare a new class or enumeration. **If the `auto` *type-specifier* appears in the *type-specifier-seq*, the *type-specifier-seq* shall contain no other *type-specifiers* except *cv-qualifiers*, and the type of the identifier being declared is deduced from the *assignment-expression* as described in ([dcl.spec.auto]).**

9.2. Class member

Paragraph 6 should be:

> **A member shall not be declared with auto, extern or register storage class.** ~~A member shall not be declared auto, extern, or register.~~

4 Acknowledgments

We are grateful to Jeremy Siek, Douglas Gregor, Jeremiah Willcock, Gary Powell, Mat Marcus, Daveed Vandevoorde, David Abrahams, Andreas Hommel, Peter Dimov, and Paul Mensonides for their valuable input in preparing this proposal. Clearly, this proposal builds on input from members of the EWG as expressed in face-to-face meetings and reflector messages. The wording has been significantly improved as the result of careful, and repeated, reading by the members of the CWG.