Deducing the type of variable from its initializer expression

Programming Language C++
Document no: N1721=04-0161

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

October 21, 2004

1 Introduction

This document is the subset of the document N1705=04-0145, proposing the allowed uses of ***auto*** which were unanimously approved by the evolution group meeting on Oct 19h, 2004.

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 basic 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*** 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

auto z4[] = fii();        // z4 : A*
```

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 ***&***.

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;
```

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 7.1.5.1 Type specifiers [dcl.type.simple]

Add to the paragraph 1

— `auto` can either be a storage class specifier, or a simple type specifier. `auto` can be combined with any other type specifier, in which case it is treated as a storage class specifier. If the *decl-specifier-sequence* contains no type specifer other than `auto`, then the following restrictions apply to the *decl-specifier-sequence*:

- It must be followed by exactly one *init-declarator*, and this *init-declarator* must have a non-empty *initializer* of either of the following two forms:

  = *initializer−clause*
  (*initializer−clause*)

- The only other allowed *decl-specifiers* are *cv-qualifiers* and the storage class specifier `static`.

[*Example:* The following are valid declarations:

```
auto x = 5;
const auto *v = expr;
static auto y = 0.0;
static auto int z; // invalid, auto treated as a storage class specifier
auto int r; // ok
```

— *end example*]

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

Section 8.3 Meaning of declarators [dcl.meaning]

New paragraph after paragraph 1:

If *decl-specifier-sequence* contains the *simple-type-specifier* `auto`, the declarator is required to declare an object and to specify an initial value; the type of the 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 an occurrence of `auto`, a *type scheme*. A type scheme yields a type when the 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;
const auto& p = f();
```

the type specifiers `int unsigned` determine the type “`unsigned int`”, and the type specifier `const auto` determines the type scheme “`const auto`” ([dcl.type.simple]).]

Section 8.3.1 Pointers [dcl.ptr]

Change the first paragraph to:

In a declaration `T D` where `D` has the form

`*` $cv\!-\!qualifier\!-\!seq_{opt}$ `D1`

and the type, or type scheme, of the identifier in the declaration `T D1` is *derived-declarator-type-list* `T`,then the type, or type scheme, of the identifier of `D` is *derived-declarator-type-list cv-qualifier-seq* pointer to `T`. The cv-qualifiers apply to the pointer and not to the object pointed to.

The change to this paragraph was the addition of the “or type scheme” in two places. Similar changes are needed to Sections 8.3.2–5 discuss how references, arrays, and function types in the declarator propagate to the type of the *declarator-id*. 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 the declared 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 `U` from `T` by replacing the occurrence of `auto` with a new invented type template parameter t. Define a function template as follows:

```
template <class t>
void __f(U __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 template argument deduction would fail, the declaration is ill-formed.

[*Example:*

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

The type scheme is `const 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 <class t> void __f(const t& 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.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*]

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.