**Type deduction: auto and decltype**

When a new variable is initialized, the compiler can figure out what the type of the variable is automatically by the initializer. For this, it suffices to use auto as the type specifier for the variable:

|  |  |  |
| --- | --- | --- |
| 1 2 | int foo = 0;  auto bar = foo; // the same as: int bar = foo; |  |

Here, bar is declared as having an auto type; therefore, the type of bar is the type of the value used to initialize it: in this case it uses the type of foo, which is int.  
  
Variables that are not initialized can also make use of type deduction with the decltype specifier:

|  |  |  |
| --- | --- | --- |
| 1 2 | int foo = 0;  decltype(foo) bar; // the same as: int bar; |  |

Here, bar is declared as having the same type as foo.  
  
auto and decltype are powerful features recently added to the language. But the type deduction features they introduce are meant to be used either when the type cannot be obtained by other means or when using it improves code readability. The two examples above were likely neither of these use cases. In fact they probably decreased readability, since, when reading the code, one has to search for the type of foo to actually know the type of bar.

**auto specifier (since C++11)**

From cppreference.com

For variables, specifies that the type of the variable that is being declared will be automatically deduced from its initializer. For functions, specifies that the return type is a trailing return type or will be deduced from its return statements (since C++14). for non-type template parameters, specifies that the type will be deduced from the argument (since C++17)

**[**[**edit**](http://en.cppreference.com/mwiki/index.php?title=cpp/language/auto&action=edit&section=1)**]**

**Syntax**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | | |
| auto *variable* *initializer* | (1) | (since C++11) |  |  |  |  |  |  |  |
|  | | | | | | | | | |
| auto *function* -> *return type* | (2) | (since C++11) |  |  |  |  |  |  |  |
|  | | | | | | | | | |
| auto *function* | (3) | (since C++14) |  |  |  |  |  |  |  |
|  | | | | | | | | | |
| decltype(auto) *variable* *initializer* | (4) | (since C++14) |  |  |  |  |  |  |  |
|  | | | | | | | | | |
| decltype(auto) *function* | (5) | (since C++14) |  |  |  |  |  |  |  |
|  | | | | | | | | | |
| auto **::** | (6) | (concepts TS) |  |  |  |  |  |  |  |
|  | | | | | | | | | |
| *cv*(optional) auto *ref*(optional) *parameter* | (7) | (since C++14) |  |  |  |  |  |  |  |
|  | | | | | | | | | |
| **template** **<** auto *Parameter* **>** | (8) | (since C++17) |  |  |  |  |  |  |  |
|  | | | | | | | | | |
| *cv*(optional) auto *ref*(optional) **[** *identifier-list* **]** *initializer* **;** | (9) | (since C++17) |  |  |  |  |  |  |  |
|  | | | | | | | | | |

**[**[**edit**](http://en.cppreference.com/mwiki/index.php?title=cpp/language/auto&action=edit&section=2)**] Explanation**

1) When declaring variables in block scope, in namespace scope, in initialization statements of for loops, etc., the keyword auto may be used as the type specifier.

Once the type of the initializer has been determined, the compiler determines the type that will replace the keyword auto using the rules for [template argument deduction](http://en.cppreference.com/w/cpp/language/template_argument_deduction) from a function call (see [template argument deduction#Other contexts](http://en.cppreference.com/w/cpp/language/template_argument_deduction#Other_contexts) for details). The keyword auto may be accompanied by modifiers, such as const or &, which will participate in the type deduction. For example, given const auto& i = expr;, the type of i is exactly the type of the argument u in an imaginary template template<class U> void f(const U& u) if the function call f(expr) was compiled. Therefore, auto&& may be deduced either as an lvalue reference or rvalue reference according to the initializer, which is used in range-based for loop.

If auto is used to declare multiple variables, the deduced types must match. For example, the declaration auto i = 0, d = 0.0; is ill-formed, while the declaration auto i = 0, \*p = &i; is well-formed and the auto is deduced as int.

2) In a [function declaration](http://en.cppreference.com/w/cpp/language/function) that uses the trailing return type syntax, the keyword auto does not perform automatic type detection. It only serves as a part of the syntax.

3) In a [function declaration](http://en.cppreference.com/w/cpp/language/function) that does not use the trailing return type syntax, the keyword auto indicates that the return type will be deduced from the operand of its [return statement](http://en.cppreference.com/w/cpp/language/return) using the rules for [template argument deduction](http://en.cppreference.com/w/cpp/language/template_argument_deduction#Other_contexts).

4) If the declared type of the variable is decltype(auto), the keyword auto is replaced with the expression (or expression list) of its initializer, and the actual type is deduced using the rules for [decltype](http://en.cppreference.com/w/cpp/language/decltype).

5) If the return type of the function is declared decltype(auto), the keyword auto is replaced with the operand of its return statement, and the actual return type is deduced using the rules for [decltype](http://en.cppreference.com/w/cpp/language/decltype).

6) A nested-name-specifier of the form auto:: is a placeholder that is replaced by a class or enumeration type following the rules for [constrained type](http://en.cppreference.com/w/cpp/language/constraints) placeholder deduction.

7) A parameter declaration in a [lambda expression](http://en.cppreference.com/w/cpp/language/lambda). (since C++14) A [function parameter](http://en.cppreference.com/w/cpp/language/function#Parameter_list) declaration. (concepts TS)

8) If a [template parameter](http://en.cppreference.com/w/cpp/language/template_parameters) is declared auto, its type is deduced from the corresponding argument.

9) A [structured binding declaration](http://en.cppreference.com/w/cpp/language/declarations#Structured_binding_declaration)

**[**[**edit**](http://en.cppreference.com/mwiki/index.php?title=cpp/language/auto&action=edit&section=3)**] Notes**

Until C++11, auto had the semantic of a [storage duration specifier](http://en.cppreference.com/w/cpp/language/storage_duration).

Mixing auto variables and functions in one declaration, as in auto f() -> int, i = 0; is not allowed.

**[**[**edit**](http://en.cppreference.com/mwiki/index.php?title=cpp/language/auto&action=edit&section=4)**] Example**

The example showing output using one of the implementations where typeinfo::name prints full type names; filter through c++filt -t if using gcc or similar.

**Run this code**

#include <iostream>

#include <cmath>

#include <typeinfo>

template<class T, class U>

auto add(T t, U u) -> decltype(t + u) // the return type is the type of operator+(T, U)

{

return t + u;

}

auto get\_fun(int arg) -> double (\*)(double) // same as: double (\*get\_fun(int))(double)

{

switch (arg)

{

case 1: return [std::fabs](http://en.cppreference.com/w/cpp/numeric/math/fabs);

case 2: return [std::sin](http://en.cppreference.com/w/cpp/numeric/math/sin);

default: return [std::cos](http://en.cppreference.com/w/cpp/numeric/math/cos);

}

}

int main()

{

auto a = 1 + 2;

[std::cout](http://en.cppreference.com/w/cpp/io/cout) << "type of a: " << typeid(a).name() << '\n';

auto b = add(1, 1.2);

[std::cout](http://en.cppreference.com/w/cpp/io/cout) << "type of b: " << typeid(b).name() << '\n';

auto c = {1, 2};

[std::cout](http://en.cppreference.com/w/cpp/io/cout) << "type of c: " << typeid(c).name() << '\n';

auto my\_lambda = [](int x) { return x + 3; };

[std::cout](http://en.cppreference.com/w/cpp/io/cout) << "my\_lambda: " << my\_lambda(5) << '\n';

auto my\_fun = get\_fun(2);

[std::cout](http://en.cppreference.com/w/cpp/io/cout) << "type of my\_fun: " << typeid(my\_fun).name() << '\n';

[std::cout](http://en.cppreference.com/w/cpp/io/cout) << "my\_fun: " << my\_fun(3) << '\n';

// auto int x; // error as of C++11: "auto" is no longer a storage-class specifier

}

Possible output:

type of a: int

type of b: double

type of c: std::initializer\_list<int>

my\_lambda: 8

type of my\_fun: double (\*)(double)

my\_fun: 0.14112