

10.8 — Type deduction for objects using the auto keyword

👤 ALEX¹ ⌚ SEPTEMBER 25, 2024

There's a subtle redundancy lurking in this simple variable definition:

```
1 | double d{ 5.0 };
```

In C++, we are required to provide an explicit type for all objects. Thus, we've specified that variable `d` is of type `double`.

However, the literal value `5.0` used to initialize `d` also has type `double` (implicitly determined via the format of the literal).

Related content

We discuss how literal types are determined in lesson [5.2 -- Literals](https://www.learncpp.com/cpp-tutorial/literals/) (<https://www.learncpp.com/cpp-tutorial/literals/>)².

In cases where we want a variable and its initializer to have the same type, we're effectively providing the same type information twice.

Type deduction for initialized variables

Type deduction (also sometimes called **type inference**) is a feature that allows the compiler to deduce the type of an object from the object's initializer. When defining a variable, type deduction can be invoked by using the `auto` keyword can be used in place of the variable's type:

```
1 | int main()  
2 | {  
3 |     auto d { 5.0 }; // 5.0 is a double literal, so d will be deduced as a double  
4 |     auto i { 1 + 2 }; // 1 + 2 evaluates to an int, so i will be deduced as an int  
5 |     auto x { i }; // i is an int, so x will be deduced as an int  
6 |  
7 |     return 0;  
8 | }
```

In the first case, because `5.0` is a double literal, the compiler will deduce that variable `d` should be of type `double`. In the second case, the expression `1 + 2` yields an `int` result, so variable `i` will be of type `int`. In the third case, `i` was previously deduced to be of type `int`, so `x` will also be deduced to be of type `int`.

Warning

Prior to C++17, `auto d{ 5.0 };` would deduce `d` to be of type `std::initializer_list<double>` rather than `double`. This was fixed in C++17, and many compilers (such as gcc and Clang) have backported this change to previous language standards.

If you are using C++14 or older, and the above example doesn't compile on your compiler, use copy initialization with `auto` instead (`auto d = 5.0`).

Because function calls are valid expressions, we can even use type deduction when our initializer is a non-void function call:

```
1 | int add(int x, int y)
2 | {
3 |     return x + y;
4 | }
5 |
6 | int main()
7 | {
8 |     auto sum { add(5, 6) }; // add() returns an int, so sum's type will be deduced as
9 |     an int
10 |
11 |     return 0;
    }
```

The `add()` function returns an `int` value, so the compiler will deduce that variable `sum` should have type `int`.

Literal suffixes can be used in combination with type deduction to specify a particular type:

```
1 | int main()
2 | {
3 |     auto a { 1.23f }; // f suffix causes a to be deduced to float
4 |     auto b { 5u };    // u suffix causes b to be deduced to unsigned int
5 |
6 |     return 0;
7 | }
```

Variables using type deduction may also use other specifiers/qualifiers, such as `const` or `constexpr`:

```
1 | int main()
2 | {
3 |     int a { 5 };          // a is an int
4 |
5 |     const auto b { 5 };   // b is a const int
6 |     constexpr auto c { 5 }; // c is a constexpr int
7 |
8 |     return 0;
9 | }
```

Type deduction must have something to deduce from

Type deduction will not work for objects that either do not have initializers or have empty initializers. It also will not work when the initializer has type `void` (or any other incomplete type). Thus, the following is not valid:

```

1  #include <iostream>
2
3  void foo()
4  {
5  }
6
7  int main()
8  {
9      auto a;           // The compiler is unable to deduce the type of a
10     auto b { };       // The compiler is unable to deduce the type of b
11     auto c { foo() }; // Invalid: c can't have type incomplete type void
12
13     return 0;
14 }

```

Although using type deduction for fundamental data types only saves a few (if any) keystrokes, in future lessons we will see examples where the types get complex and lengthy (and in some cases, can be hard to figure out). In those cases, using `auto` can save a lot of typing (and typos).

Related content

The type deduction rules for pointers and references are a bit more complex. We discuss these in [12.14 -- Type deduction with pointers, references, and const](https://www.learncpp.com/cpp-tutorial/type-deduction-with-pointers-references-and-const/) (<https://www.learncpp.com/cpp-tutorial/type-deduction-with-pointers-references-and-const/>)³.

Type deduction drops `const` from the deduced type

In most cases, type deduction will drop the `const` from deduced types. For example:

```

1  int main()
2  {
3      const int a { 5 }; // a has type const int
4      auto b { a };     // b has type int (const dropped)
5
6      return 0;
7  }

```

In the above example, `a` has type `const int`, but when deducing a type for variable `b` using `a` as the initializer, type deduction deduces the type as `int`, not `const int`.

If you want a deduced type to be `const`, you must supply the `const` yourself as part of the definition:

```

1  int main()
2  {
3      const int a { 5 }; // a has type const int
4      const auto b { a }; // b has type const int (const dropped but reapplied)
5
6
7      return 0;
8  }

```

In this example, the type deduced from `a` will be `int` (the `const` is dropped), but because we've re-added a `const` qualifier during the definition of variable `b`, variable `b` will have type `const int`.

Type deduction for string literals

For historical reasons, string literals in C++ have a strange type. Therefore, the following probably won't work as expected:

```
1 | auto s { "Hello, world" }; // s will be type const char*, not std::string
```

If you want the type deduced from a string literal to be `std::string` or `std::string_view`, you'll need to use the `s` or `sv` literal suffixes (introduced in lessons [5.7 -- Introduction to std::string](https://www.learncpp.com/cpp-tutorial/introduction-to-stdstring/) (<https://www.learncpp.com/cpp-tutorial/introduction-to-stdstring/>)⁴ and [5.8 -- Introduction to std::string_view](https://www.learncpp.com/cpp-tutorial/introduction-to-stdstring_view/) (https://www.learncpp.com/cpp-tutorial/introduction-to-stdstring_view/)⁵):

```
1 | #include <string>
2 | #include <string_view>
3 |
4 | int main()
5 | {
6 |     using namespace std::literals; // easiest way to access the s and sv suffixes
7 |
8 |     auto s1 { "goo"s }; // "goo"s is a std::string literal, so s1 will be deduced as
9 |     a std::string
10 |     auto s2 { "moo"sv }; // "moo"sv is a std::string_view literal, so s2 will be
11 |     deduced as a std::string_view
12 |
13 |     return 0;
14 | }
```

But in such cases, it may be better to not use type deduction.

Type deduction and constexpr

Because `constexpr` is not part of the type system, it cannot be deduced as part of type deduction. However, a `constexpr` variable is implicitly `const`, and this `const` will be dropped during type deduction (and can be readded if desired):

```
1 | int main()
2 | {
3 |     constexpr double a { 3.4 }; // a has type const double (constexpr not part of
4 |     type, const is implicit)
5 |
6 |     auto b { a };                // b has type double (const dropped)
7 |     const auto c { a };          // c has type const double (const dropped but
8 |     reapplied)
9 |     constexpr auto d { a };      // d has type const double (const dropped but
10 |     implicitly reapplied by constexpr)
11 |
12 |     return 0;
13 | }
```

Type deduction benefits and downsides

Type deduction is not only convenient, but also has a number of other benefits.

First, if two or more variables are defined on sequential lines, the names of the variables will be lined up, helping to increase readability:

```

1 // harder to read
2 int a { 5 };
3 double b { 6.7 };
4
5 // easier to read
6 auto c { 5 };
7 auto d { 6.7 };

```

Second, type deduction only works on variables that have initializers, so if you are in the habit of using type deduction, it can help avoid unintentionally uninitialized variables:

```

1 int x; // oops, we forgot to initialize x, but the compiler may not complain
2 auto y; // the compiler will error out because it can't deduce a type for y

```

Third, you are guaranteed that there will be no unintended performance-impacting conversions:

```

1 std::string_view getString(); // some function that returns a std::string_view
2
3 std::string s1 { getString() }; // bad: expensive conversion from std::string_view to
  std::string (assuming you didn't want this)
4 auto s2 { getString() }; // good: no conversion required

```

Type deduction also has a few downsides.

First, type deduction obscures an object's type information in the code. Although a good IDE should be able to show you the deduced type (e.g. when hovering a variable), it's still a bit easier to make type-based mistakes when using type deduction.

For example:

```

1 auto y { 5 }; // oops, we wanted a double here but we accidentally provided an int
  literal

```

In the above code, if we'd explicitly specified `y` as type `double`, `y` would have been a double even though we accidentally provided an int literal initializer. With type deduction, `y` will be deduced to be of type `int`.

Here's another example:

```

1 #include <iostream>
2
3 int main()
4 {
5     auto x { 3 };
6     auto y { 2 };
7
8     std::cout << x / y << '\n'; // oops, we wanted floating point division here
9
10    return 0;
11 }

```

In this example, it's less clear that we're getting an integer division rather than a floating-point division.

Similar cases occur when a variable is `unsigned`. Since we don't want to mix signed and unsigned values, explicitly knowing that a variable has an unsigned type is generally something that shouldn't be obscured.

Second, if the type of an initializer changes, the type of a variable using type deduction will also change, perhaps unexpectedly. Consider:

```
1 | auto sum { add(5, 6) + gravity };
```

If the return type of `add` changes from `int` to `double`, or `gravity` changes from `int` to `double`, `sum` will also change type from `int` to `double`.

Overall, the modern consensus is that type deduction is generally safe to use for objects, and that doing so can help make your code more readable by de-emphasizing type information so the logic of your code stands out better.

Best practice

Use type deduction for your variables when the type of the object doesn't matter.

Favor an explicit type when you require a specific type that differs from the type of the initializer, or when your object is used in a context where making the type obvious is useful.

Author's note

In future lessons, we'll continue to use explicit types instead of type deduction when we feel showing the type information is helpful to understanding a concept or example.



Next lesson

10.9 [Type deduction for functions](#)



[Back to table of contents](#)



Previous lesson

10.7 [Typedefs and type aliases](#)



B

U

URL

INLINE CODE

C++ CODE BLOCK

HELP!

Leave a comment...

 Name*


@ Email* | ?

Notify me about replies:



POST COMMENT

 Find a mistake? Leave a comment above!?

 Avatars from <https://gravatar.com/>¹² are connected to your provided email address.

172 COMMENTS

Newest ▼



Tabi

🕒 May 30, 2025 5:40 am PDT

Great post! Just a small note: the phrase "can be invoked by using the auto keyword can be used" has a repeated "can be". Maybe a quick edit could make it clearer. Hope that helps!

👍 2 ➡ Reply



JhnDevil

🕒 February 11, 2025 1:26 am PST

Nice

 Last edited 4 months ago by JhnDevil

👍 8 ➡ Reply



NordicPeace

🕒 December 12, 2024 8:10 am PST

```
1 using namespace std::literals;
2
3 auto d{ 12.232 };
4 cout << typeid(d).name() << endl; // Result : Double
5
6 auto x{ 33 };
7 cout << typeid(x).name() << endl; // Result : int
8 auto y{ "eden"sv };
9 cout << typeid(y).name() << endl; /*
10
11 class std::basic_string_view<char,struct std::char_traits<char> >*/
```

Why the last one doesn't give us just "string_view" and What is class std:: basic_string_view<char, struct std :: char_trait <char> ?

👍 1 ➡ Reply



Alex Author

🔄 Reply to [NordicPeace](#)¹³ ⌚ December 16, 2024 8:23 pm PST

Because `string_view` is actually an alias for `std::basic_string_view<char, struct std::char_traits<char>>`. You can see that here:
https://en.cppreference.com/w/cpp/string/basic_string_view.

👍 1 ➡ Reply



NordicPeace

🔄 Reply to [Alex](#)¹⁴ ⌚ December 16, 2024 11:16 pm PST

There are a hell lot of aliases that exist in c++ XD, btw the site is broken for some reason and has no text on it.

👍 0 ➡ Reply



Skyst1ke

⌚ November 13, 2024 11:01 am PST

Is it better to do:

```
unsigned int x { 5 };
```

Or:

```
auto x { 5u };
```

👍 0 ➡ Reply



Alex Author

🔄 Reply to [Skyst1ke](#)¹⁵ ⌚ November 15, 2024 1:29 pm PST

Either is fine.

👍 1 ➡ Reply



tarik2306

⌚ October 13, 2024 4:43 am PDT

is it possible to change the "type deduction" settings? Can I make it always return long or short instead of int?

👍 0 ➡ Reply



Alex Author

🔄 Reply to [tarik2306](#)¹⁶ ⌚ October 16, 2024 2:53 pm PDT

No. If you want it to return a long or short, your initializer needs to be a long or short.



TheSettlers

🕒 June 9, 2024 3:22 am PDT

Probably off-topic:

```
int variableName {100}; // declaration and initialization
```

```
variableName = 95; // 1 ?
```

```
variableName = {95}; // 2 ?
```

Which is the best option to use to update the data in the variable?

👍 1 ➡ Reply



Alex Author

🔗 Reply to [TheSettlers](#)¹⁷ 🕒 June 9, 2024 9:11 pm PDT

#1

👍 2 ➡ Reply



Hamza

🕒 May 16, 2024 8:19 am PDT

Hi Alex, when I was looking online about declaring variables with auto, I came across this article:

<https://stackoverflow.com/questions/29890918/why-is-direct-list-initialization-with-auto-considered-bad-or-not-preferred> (<https://stackoverflow.com/questions/29890918/why-is-direct-list-initialization-with-auto-considered-bad-or-not-preferred>)¹⁸

Showing that using direct list initialization with auto does not work in all cases.

The article above also has a link to this article which somewhat covers declaring variables with auto:

<https://herbsutter.com/2013/08/12/gotw-94-solution-aaa-style-almost-always-auto/> (<https://herbsutter.com/2013/08/12/gotw-94-solution-aaa-style-almost-always-auto/>)¹⁹

But I've also seen a lot of C++ code (the C++ commercial codebase I'm in) that simply doesn't use direct list initialization with auto at all, even when it would work fine, and opt for copy initialization instead:

```
1 | auto myBigClass = std::make_unique<BigClass>();
2 |
3 | // Why not?
4 | auto myBigClass{std::make_unique<BigClass>()};
```

I've also seen instances of auto being used like follows:

```
1 | // ... in some unit test code...
2 | auto fooId = uint64_t(20);
3 |
4 | // Is the above overkill, or is it expressing some intent I'm unaware of?
5 | uint64_t fooId{20};
```

It seems the current consensus is auto should not be used with direct list initialization at all. Is it because auto just has too many weird quirks with list initialization and copy initialization doesn't?

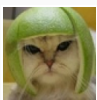
Personally I find it conflicting how the main recommended way to initialize variables is with direct list init, except with auto, where its instead copy initialization. Then here, you use auto with direct list initialization instead.

What are your thoughts on using auto with direct list initialization when compared to auto with copy initialization?

(I should mention that I'm currently an intern learning C++, which is why I have access to this particular codebase. I have no strong opinions on this topic yet haha)

 Last edited 1 year ago by Hamza

 0  Reply



Alex Author

 Reply to [Hamza](#)²⁰  May 16, 2024 12:17 pm PDT

C++17 fixed the main issue, so now `auto var { expr }` will result in the same type deduction as `auto var = expr`.

IMO, it's fine to use auto with direct list init in C++17.

I'll update the article to mention this defect pre-C++17.

 3  Reply



Jonh

 March 18, 2024 5:39 am PDT

Alex, I know it's better to develop our code for readability, but does use 'auto' don't degrade the performance? Could you talk about this or indicate a link? Thanks!

 1  Reply



Alex Author

 Reply to [Jonh](#)²¹  March 18, 2024 11:33 am PDT

All `auto` does is ask the compiler to infer what type something is rather than having us explicitly specify it. That will have a slight compile-time performance cost (but nothing to worry about), but no runtime performance cost (as the compiled code should be identical).

 4  Reply



Jankat

 March 16, 2024 2:27 pm PDT

It seems like it kinda makes stuff harder to read if overused. When reading code I want to be able to see what dtypes are used with a glance instead of reading each initialization thoroughly.

Tho I do like how it works when calling functions a LOT. Helps abstract stuff. I didn't like python not

having dtypes because I couldn't tell where or when a variable is initialized it could benefit a lot from copying this design.

 Last edited 1 year ago by Jankat

 2  Reply



Ajit Mali

 February 22, 2024 1:30 am PST

Yoo! Something new, I never new that cpp also has this

 Last edited 1 year ago by Ajit Mali

 2  Reply

Links

1. <https://www.learncpp.com/author/Alex/>
2. <https://www.learncpp.com/cpp-tutorial/literals/>
3. <https://www.learncpp.com/cpp-tutorial/type-deduction-with-pointers-references-and-const/>
4. <https://www.learncpp.com/cpp-tutorial/introduction-to-stdstring/>
5. https://www.learncpp.com/cpp-tutorial/introduction-to-stdstring_view/
6. <https://www.learncpp.com/cpp-tutorial/type-deduction-for-functions/>
7. <https://www.learncpp.com/>
8. <https://www.learncpp.com/cpp-tutorial/typedefs-and-type-aliases/>
9. <https://www.learncpp.com/type-deduction-for-objects-using-the-auto-keyword/>
10. <https://www.learncpp.com/cpp-tutorial/chapter-8-summary-and-quiz/>
11. <https://www.learncpp.com/cpp-tutorial/range-based-for-loops-for-each/>
12. <https://gravatar.com/>
13. <https://www.learncpp.com/cpp-tutorial/type-deduction-for-objects-using-the-auto-keyword/#comment-605184>
14. <https://www.learncpp.com/cpp-tutorial/type-deduction-for-objects-using-the-auto-keyword/#comment-605336>
15. <https://www.learncpp.com/cpp-tutorial/type-deduction-for-objects-using-the-auto-keyword/#comment-604135>
16. <https://www.learncpp.com/cpp-tutorial/type-deduction-for-objects-using-the-auto-keyword/#comment-603030>
17. <https://www.learncpp.com/cpp-tutorial/type-deduction-for-objects-using-the-auto-keyword/#comment-598167>
18. <https://stackoverflow.com/questions/29890918/why-is-direct-list-initialization-with-auto-considered-bad-or-not-preferred>
19. <https://herbsutter.com/2013/08/12/gotw-94-solution-aaa-style-almost-always-auto/>

20. <https://www.learncpp.com/cpp-tutorial/type-deduction-for-objects-using-the-auto-keyword/#comment-597158>
21. <https://www.learncpp.com/cpp-tutorial/type-deduction-for-objects-using-the-auto-keyword/#comment-594816>
22. <https://g.ezoic.net/privacy/learncpp.com>