10.8 — Type deduction for objects using the auto keyword

learncpp.com/cpp-tutorial/type-deduction-for-objects-using-the-auto-keyword/

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

```
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.

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. To use type deduction with variables, the `auto` keyword is used in place of the variable’s type:

```
int main()
{
    auto d{ 5.0 }; // 5.0 is a double literal, so d will be type double
    auto i{ 1 + 2 }; // 1 + 2 evaluates to an int, so i will be type int
    auto x { i }; // i is an int, so x will be type int too

    return 0;
}
```

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`.

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

```
int add(int x, int y)
{
    return x + y;
}

int main()
{
    auto sum { add(5, 6) }; // add() returns an int, so sum's type will be deduced to
int

    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:

```
int main()
{
    auto a { 1.23f }; // f suffix causes a to be deduced to float
    auto b { 5u };    // u suffix causes b to be deduced to unsigned int

    return 0;
}
```

Type deduction will not work for objects that 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:

```
#include <iostream>

void foo()
{
}

int main()
{
    auto x;          // The compiler is unable to deduce the type of x
    auto y{ };       // The compiler is unable to deduce the type of y
    auto z{ foo() }; // z can't have type void, so this is invalid

    return 0;
}
```

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.

Type deduction drops const / constexpr qualifiers

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

```
int main()
{
    const int x { 5 }; // x has type const int
    auto y { x };      // y will be type int (const is dropped)

    return 0;
}
```

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

If you want a deduced type to be const or constexpr, you must supply the const or constexpr yourself. To do so, simply use the `const` or `constexpr` keyword in conjunction with the `auto` keyword:

```
int main()
{
    const int x { 5 };  // x has type const int (compile-time const)
    auto y { x };       // y will be type int (const is dropped)

    constexpr auto z { x }; // z will be type constexpr int (constexpr is reapplied)

    return 0;
}
```

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

```
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 lesson 5.2 -- Literals):

```
#include <string>
#include <string_view>

int main()
{
    using namespace std::literals; // easiest way to access the s and sv suffixes

    auto s1 { "goo"s };  // "goo"s is a std::string literal, so s1 will be deduced as
a std::string
    auto s2 { "moo"sv }; // "moo"sv is a std::string_view literal, so s2 will be
deduced as a std::string_view

    return 0;
}
```

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

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:

```
// harder to read
int a { 5 };
double b { 6.7 };

// easier to read
auto c { 5 };
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:

```
int x; // oops, we forgot to initialize x, but the compiler may not complain
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:

```
std::string_view getString();   // some function that returns a std::string_view

std::string s1 { getString() }; // bad: expensive conversion from std::string_view to
std::string (assuming you didn't want this)
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:

```
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:

```
#include <iostream>

int main()
{
     auto x { 3 };
     auto y { 2 };

     std::cout << x / y << '\n'; // oops, we wanted floating point division here

     return 0;
}
```

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:

```
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.