8.7 — Type deduction for objects using the auto keyword

ALEX SEPTEMBER 28, 2021

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

```
double d{ 5.0 };
```

Because C++ is a strongly-typed language, 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 4.13 -- 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, the `auto` keyword is used in place of the variable's type:

```cpp
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 function call:

```cpp
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`.

Type deduction will not work for objects that do not have initializers or empty initializers. Thus, the following is not valid:

```cpp
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

    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).

Type deduction drops const qualifiers

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

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

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, you must supply the const yourself. To do so, simply use the `const` keyword in conjunction with the `auto` keyword:

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

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

> **For advanced readers**
>
> Type deduction will not drop the const qualifier for pointers to const values, such as types deduced from C-style string literals.

Type deduction drops references

> **For advanced readers**
>
> We haven't covered references yet, but type deduction will also drop references.
>
> For example, if you use type deduction with an initializer of type int&, the deduced type will be “int”, not “int&”.
>
> ```
> int x{ 5 }; // x is a normal int
> int& y{ x }; // y is an int& reference
> auto z{ y }; // z will be an "int", not an "int &" because references are dropped
> ```
>
> You can ensure a deduced type is a reference type by using `auto&` instead of `auto`.
>
> ```
> int x{ 5 }; // x is a normal int
> auto& y{ x }; // type deduced is "int", but we've provided an &, so y will be an "int&" reference
> ```
>
> You can also deduce a const reference by using `const auto&`

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:

```
double x { 5 }; // bad: implicitly converts 5 from an int to a double
auto y { 5 }; // good: y is an int (hopefully that's what you wanted) and no conversion takes place
```

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; // 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.

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 types from int to double.

For advanced readers

Third, because type deduction drops references, if you use “auto” when you should be using “auto&”, your code may not perform as well, or may even not work correctly.

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, unless you need to commit to a specific type.

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
8.8 Type deduction for functions

Back to table of contents

Previous lesson
8.6 Typedefs and type aliases

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ cod

Name*

Email*

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

Notify me about replies:

POST COMMENT

DP N N FOUT

Newest

X

X