10.9 — Type deduction for functions

Consider the following program:

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

When this function is compiled, the compiler will determine that `x + y` evaluates to an `int`, then ensure that type of the return value matches the declared return type of the function (or that the return value type can be converted to the declared return type).

Since the compiler already has to deduce the return type from the return statement, in C++14, the `auto` keyword was extended to do function return type deduction. This works by using the `auto` keyword in place of the function's return type.

For example:

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

Because the return statement is returning an `int` value, the compiler will deduce that the return type of this function is `int`.

When using an `auto` return type, all return statements within the function must return values of the same type, otherwise an error will result. For example:

```
auto someFcn(bool b)
{
    if (b)
        return 5; // return type int
    else
        return 6.7; // return type double
}
```

In the above function, the two return statements return values of different types, so the compiler will give an error.

If such a case is desired for some reason, you can either explicitly specify a return type for your function (in which case the compiler will try to implicitly convert any non-matching return expressions to the explicit return type), or you can explicitly convert all of your return

statements to the same type. In the example above, the latter could be done by changing `5` to `5.0`, but `static_cast` can also be used for non-literal types.

A major downside of functions that use an `auto` return type is that such functions must be fully defined before they can be used (a forward declaration is not sufficient). For example:

```
#include <iostream>

auto foo();

int main()
{
    std::cout << foo() << '\n'; // the compiler has only seen a forward declaration
at this point

    return 0;
}

auto foo()
{
    return 5;
}
```

On the author's machine, this gives the following compile error:

```
error C3779: 'foo': a function that returns 'auto' cannot be used before it is
defined.
```

This makes sense: a forward declaration does not have enough information for the compiler to deduce the function's return type. This means normal functions that return `auto` are typically only callable from within the file in which they are defined.

Unlike type deduction for objects, there isn't as much consensus on best practices for function return type deduction. When using type deduction with objects, the initializer is always present as part of the same statement, so it's usually not overly burdensome to determine what type will be deduced. With functions, that is not the case -- when looking at a function's prototype, there is no context to help indicate what type the function returns. A good programming IDE should make clear what the deduced type of the function is, but in absence of having that available, a user would actually have to dig into the function body itself to determine what type the function returned. The odds of mistakes being made are higher. And the inability for such functions to be forward declared limits their usefulness in multi-file programs.

Best practice

Favor explicit return types over function return type deduction for normal functions.

Trailing return type syntax

The `auto` keyword can also be used to declare functions using a **trailing return syntax**, where the return type is specified after the rest of the function prototype.

Consider the following function:

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

Using the trailing return syntax, this could be equivalently written as:

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

In this case, `auto` does not perform type deduction -- it is just part of the syntax to use a trailing return type.

Why would you want to use this?

One nice thing is that it makes all of your function names line up:

```
auto add(int x, int y) -> int;
auto divide(double x, double y) -> double;
auto printSomething() -> void;
auto generateSubstring(const std::string &s, int start, int len) -> std::string;
```

The trailing return syntax is also required for some advanced features of C++, such as lambdas (which we cover in lesson 20.6 -- Introduction to lambdas (anonymous functions)).

For now, we recommend the continued use of the traditional function return syntax except in situations that require the trailing return syntax.

Type deduction can’t be used for function parameter types

Many new programmers who learn about type deduction try something like this:

```
#include <iostream>

void addAndPrint(auto x, auto y)
{
    std::cout << x + y << '\n';
}

int main()
{
    addAndPrint(2, 3); // case 1: call addAndPrint with int parameters
    addAndPrint(4.5, 6.7); // case 2: call addAndPrint with double parameters

    return 0;
}
```

Unfortunately, type deduction doesn't work for function parameters, and prior to C++20, the above program won't compile (you'll get an error about function parameters not being able to have an auto type).

In C++20, the `auto` keyword was extended so that the above program will compile and function correctly -- however, `auto` is not invoking type deduction in this case. Rather, it is triggering a different feature called `function templates` that was designed to actually handle such cases.

Related content

We introduce function templates in lesson 11.6 -- Function templates, and discuss use of `auto` in the context of function templates in lesson 11.8 -- Function templates with multiple template types.