10.9 — Type deduction for functions

ALEX[1] DECEMBER 18, 2024

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).

Return type deduction with `auto`

Since the compiler already has to deduce the return type from the return statement (to ensure that the value can be converted to the function's declared return type), 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.

Benefits of return type deduction

The biggest advantage of return type deduction is that having the compiler deduce the function's return type negates the risk of a mismatched return type (preventing unexpected conversions).

This can be particularly useful when a function's return type is fragile (cases where return type is likely to change if the implementation changes). In such cases, being explicit about the return type means having to update all relevant return types when an impacting change is made to the implementation. If we're lucky, the compiler will error until we update the relevant return types. If we're not lucky, we'll get implicit conversions where we don't desire them.

In other cases, the return type of a function may either be long and complex, or not be that obvious. In such cases, `auto` can be used to simplify:

```
1  // let compiler determine the return type of unsigned short + char
2  auto add(unsigned short x, char y)
3  {
4      return x + y;
5  }
```

We discuss this case a bit more (and how to express the actual return type of such a function) in lesson 11.8 -- Function templates with multiple template types (https://www.learncpp.com/cpp-tutorial/function-templates-with-multiple-template-types/)[2].

Downsides of return type deduction

There are two major downsides to return type deduction:

1. Functions that use an `auto` return type must be fully defined before they can be used (a forward declaration is not sufficient). For example:

```
1  #include <iostream>
2
3  auto foo();
4
5  int main()
6  {
7      std::cout << foo() << '\n'; // the compiler has only seen a forward declaration at
8  this point
9
10     return 0;
11 }
12
13 auto foo()
14 {
15     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 def
```

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.

2. 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 type deduction for functions, that is not the case -- the function's prototype gives no indication as to what type the function actually 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. Generally we prefer to be explicit about types that are part of an interface (a function's declaration is an interface).

Unlike type deduction for objects, there isn't as much consensus on best practices for function return type deduction. We recommend generally avoiding return type deduction for functions.

> **Best practice**
>
> Prefer explicit return types over return type deduction (except in cases where the return type is unimportant, difficult to express, or fragile).

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:

```
1  int add(int x, int y)
2  {
3    return (x + y);
4  }
```

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

```
1  auto add(int x, int y) -> int
2  {
3    return (x + y);
4  }
```

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? Here are some reasons:

1. For functions with complex return types, a trailing return type can make the function easier to read:

```
#include <type_traits> // for std::common_type

std::common_type_t<int, double> compare(int, double);         // harder to read (where is the name of the function in this mess?)
auto compare(int, double) -> std::common_type_t<int, double>; // easier to read (we don't have to read the return type unless we care)
```

2. The trailing return type syntax can be used to align the names of your functions, which makes consecutive function declarations easier to read:

```
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;
```

For advanced readers

3. If we have a function whose return type must be deduced based on the type of the function parameters, a normal return type will not suffice, because the compiler has not yet seen the parameters at that point.

```
#include <type_traits>
// note: decltype(x) evaluates to the type of x

std::common_type_t<decltype(x), decltype(y)> add(int x, double y);         // Compile error: compiler hasn't seen definitions of x and y yet
auto add(int x, double y) -> std::common_type_t<decltype(x), decltype(y)>; // ok
```

4. 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)[3]).

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 (https://www.learncpp.com/cpp-tutorial/function-templates/)[4], and discuss use of `auto` in the context of function templates in lesson 11.8 -- Function templates with multiple template types (https://www.learncpp.com/cpp-tutorial/function-templates-with-multiple-template-types/)[2].

Next lesson
10.x Chapter 10 summary and quiz

5

Back to table of contents

6

Previous lesson
10.8 Type deduction for objects using the auto keyword

7

8

NordicPeace

December 12, 2024 8:52 am PST

Why don't we prefer templates over auto deductions?

0 Reply

Alex Author

Reply to NordicPeace [12] December 18, 2024 10:17 am PST

If you're talking about return types, we use them for different cases:

Return type deduction asks the compiler to deduce the return type from the body of the function. It requires the function definition be available. I updated the lesson a bit to be a little more clear about which cases we might prefer return type deduction.

Templates allow us to explicitly specify the return type as a normal type or as a template type parameter. It does not require the function definition be available.

If you're asking about templates vs abbreviated templates, use either.

1 Reply

Gabriel Radu Taranciuc

November 10, 2024 1:24 pm PST

So I was trying to compile the code examples above (I like keeping a log of everything I've learned), but this `auto add(int x, double y) -> std::common_type_t<x, y>;` just won't compile and will give me an error I also could find very little about: `error C3544: '_Ty': parameter pack expects a type template argument`

Couldn't really make rhyme or reason of it. Compiler is MSVC C++23 (tried with and without optimizations, no diff)

1 Reply

Alex Author

Reply to Gabriel Radu Taranciuc [13] November 11, 2024 2:05 pm PST

Mistake on my part. Since `x` and `y` are objects, not types, `std::common_type_t<x, y>` should be `std::common_type_t<decltype(x), decltype(y)>`. The lesson has been corrected.

3 Reply

Ardalan

March 31, 2024 7:33 am PDT

Thanks for your tutorial, why don't you write a more simple c++ programming language for primary students? Easy to learn for them

0 Reply

aki

Reply to Ardalan [14] June 11, 2024 12:30 am PDT

I'm a student and I have to say that i've never seen tutorials put together as well as this website. It's kind of insane how information this good is completely free. Universities will charge you thousands if not more for this kind of info lmao.

Last edited 1 year ago by aki

13 Reply

Alex Author

Reply to Ardalan [14] April 1, 2024 9:17 am PDT

Because I don't have enough time to do that and maintain/expand this site.

7 Reply

Ajit Mali

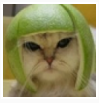
February 22, 2024 1:59 am PST

- 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).

```
#include <iostream>

void printSum(auto sum) {
    std::cout << sum << std::endl;
}

auto sum(auto n1, auto n2) {
    return n1 + n2;
}

int main() {
    printSum(sum(5, 6));
    printSum(sum(5.5, 6.5));
    return 0;
}
```

This works fine, I use GCC compiler with C++14 standard and didn't got any error, How?

0 Reply



Alex Author

Reply to Ajit Mali [15] February 23, 2024 2:27 pm PST

GCC is non-conformant in allowing access to this capability while specifying the C++14 language standard.

It does produce a warning though.

1 Reply

zls

November 5, 2023 6:32 am PST

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

Here I can use add with int and int, double and double, int and double etc. So is this better alternative to function overloading?

0 Reply

Alex Author

Reply to zls [16] November 5, 2023 2:31 pm PST

It depends. If you want `x` and `y` to have any type that will compile, then yes, this is more concise than explicitly defined functions. If you want x and y to only have specific combinations of types, then you should be define explicit overloads.

Do note that the above creates overloaded functions, it just does so implicitly.

0 Reply

Krishnakumar

June 28, 2023 5:00 pm PDT

I feel that it would help with learner motivation if we stop this chapter now, and split the rest of the content (function overloading) into a separate chapter.

This lesson provides a nice demarcation point for a chapter titled "More on data types", since it ends all **basic topics** of static typing, viz. implicit/explicit conversions, arithmetic rules, aliases and auto.

The next chapter can be titled "Function Overloading" and like all good tutorials, build nicely on the data type aspects covered in this chapter, but yet form a logically independent topic.

Learners shall feel rewarded that they managed to finish yet another chapter if we keep it to such a reasonable length.

4 Reply

Alex Author

Reply to Krishnakumar [17] June 29, 2023 12:47 pm PDT

Agreed. It's on my todo to split some of the longer chapters into multiple, shorter chapters.

1 Reply

Emeka Daniel

Reply to Alex [18] August 14, 2023 9:56 am PDT

I hope splitting the Random Numbers section from the Control Flow and Error Handling Chapter is part of your todo too :D .

0 Reply

Alex Author

Reply to Emeka Daniel [19] August 17, 2023 5:23 pm PDT

I was intending to leave randomization as part of control flow, since randomization allows for lots of simple but interesting games that exercise control flow.

1 Reply

Emeka Daniel

Reply to Alex [20] August 18, 2023 1:24 am PDT

Yh true. But could you atleast divide it into a separate compulsory chapter called 'CHAPTER R' after the control flow chapter, because it kinda of feels off going from control flow and error handling topics straight to Random Numbers, you could even add a short description of why the chapter is been taught now, instead of learning it after class types - that it is essential to the quizzes from now on, something like that. That's if all this is convenient for you, if it's not then there's no problem.

0 Reply

Swaminathan R

Reply to Emeka Daniel [21] May 2, 2024 10:11 am PDT

Random number topic after Control flow felt like Full metal jacket's first and second halves..

1 Reply

red sus

June 21, 2023 12:45 pm PDT

trailing functions generate function templates?

0 Reply

Alex Author

Reply to red sus [22] June 23, 2023 9:19 pm PDT

No, using `auto` as the type of a function parameter generates a function template in C++20.

1 Reply

red sus

June 21, 2023 12:44 pm PDT

ultra W lesson men now my function look more pro

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

ultra pro

19 Reply

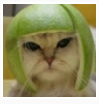
anonymous

June 8, 2023 5:17 pm PDT

Made a program to test out a function capable of taking two different types.
Don't think there's much benefit to this specific program however.

```cpp
#include <iostream>

auto add(auto x, auto y)//changes type to either int or double depending on which
input
                        //function is called
{
    return x + y;
}
int inputInt()
{
    std::cout << "Please input two integer values: ";
    int a{};
    int b{};
    std::cin >> a >> b;
    return add(a, b);
}
double inputDouble()
{
    std::cout << "Please input two double values: ";
    double a{};
    double b{};
    std::cin >> a >> b;
    return add(a, b);
}

int main()
{
    std::cout << "Select int or double (i/d): ";
    char iOrD{};
    std::cin >> iOrD;

    if (iOrD == 'i')
    {
        int answer{ inputInt() };
        std::cout << answer;
    }
    else if (iOrD == 'd')
    {
        double answer{ inputDouble() };
        std::cout << answer;
    }
    else
        std::cout << "Error, wrong input";

    return 0;
}
```

0 Reply



Alex Author

Reply to anonymous [23] June 12, 2023 9:46 am PDT

Functions with auto parameters are actually abbreviated function templates. We cover these in https://www.learncpp.com/cpp-tutorial/function-templates-with-multiple-template-types/

4 Reply

İsmail

March 15, 2023 7:53 am PDT

When programming with a language that has type definitions, how appropriate is it to use a type like auto? How correct is the usage of auto? If we use auto, wouldn't the data we are working with be disorganized and uncategorized? Isn't that bad? Where do we need auto and the trailing return type (->)?

0 Reply

Alex Author

Reply to İsmail [24] March 16, 2023 1:26 pm PDT

The lessons on auto in this chapter answer these usage questions. Auto on variables is usually fine, auto on function return types should generally be avoided except in some template function cases.

The trailing return type is an optional syntax that some developers prefer (because it left-aligns the names of a group of functions). It's also required when writing lambdas, which we cover later.

2 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/function-templates-with-multiple-template-types/
3. https://www.learncpp.com/cpp-tutorial/introduction-to-lambdas-anonymous-functions/
4. https://www.learncpp.com/cpp-tutorial/function-templates/
5. https://www.learncpp.com/cpp-tutorial/chapter-10-summary-and-quiz/
6. https://www.learncpp.com/
7. https://www.learncpp.com/cpp-tutorial/type-deduction-for-objects-using-the-auto-keyword/
8. https://www.learncpp.com/type-deduction-for-functions/
9. https://www.learncpp.com/cpp-tutorial/arithmetic-conversions/
10. https://www.learncpp.com/cpp-tutorial/function-overload-differentiation/
11. https://gravatar.com/
12. https://www.learncpp.com/cpp-tutorial/type-deduction-for-functions/#comment-605187
13. https://www.learncpp.com/cpp-tutorial/type-deduction-for-functions/#comment-604046
14. https://www.learncpp.com/cpp-tutorial/type-deduction-for-functions/#comment-595285
15. https://www.learncpp.com/cpp-tutorial/type-deduction-for-functions/#comment-593906
16. https://www.learncpp.com/cpp-tutorial/type-deduction-for-functions/#comment-589426
17. https://www.learncpp.com/cpp-tutorial/type-deduction-for-functions/#comment-582754
18. https://www.learncpp.com/cpp-tutorial/type-deduction-for-functions/#comment-582850
19. https://www.learncpp.com/cpp-tutorial/type-deduction-for-functions/#comment-585583
20. https://www.learncpp.com/cpp-tutorial/type-deduction-for-functions/#comment-585718
21. https://www.learncpp.com/cpp-tutorial/type-deduction-for-functions/#comment-585757
22. https://www.learncpp.com/cpp-tutorial/type-deduction-for-functions/#comment-582241
23. https://www.learncpp.com/cpp-tutorial/type-deduction-for-functions/#comment-581482
24. https://www.learncpp.com/cpp-tutorial/type-deduction-for-functions/#comment-578405
25. https://g.ezoic.net/privacy/learncpp.com