6.14 — The auto keyword

BY ALEX ON JULY 30TH, 2015 | LAST MODIFIED BY ALEX ON JANUARY 3RD, 2020

Prior to C++11, the auto keyword was probably the least used keyword in C++. However, C++11 and newer language standards have given new life to the auto keyword. We'll cover many of those uses here.

Consider the following statement:

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

If C++ already knows 5.0 is a double literal, why do we have to explicitly specify that d is actually a double? Wouldn't it be nice if we could tell a variable to just assume the proper type based on the value we're initializing it with?

Type inference for initialized variables

When initializing a variable, the auto keyword can be used in place of the type to tell the compiler to infer the variable's type from the initializer's type. This is called **type inference** (also sometimes called **type deduction**).

For example:

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

This also works with the return values from functions:

```
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 to int
9      return 0;
10 }
```

While using auto in place of fundamental data types only saves a few (if any) keystrokes, in future lessons we will see examples where the types get complex and lengthy. In those cases, using auto can save a lot of typing.

Type inference for functions in C++14

In C++14, the auto keyword was extended to be able to deduce a function's return type from return statements in the function body. Consider the following program:

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

Since x + y evaluates to an int, the compiler will deduce this function should have a return type of int. When using an auto return type, all return statements must return the same type, otherwise an error will result.

While this may seem neat, we recommend that this syntax be avoided for normal functions. The return type of a function is of great use in helping to document for the caller what a function is expected to return. When a specific

type isn't specified, the caller may misinterpret what type the function will return, which can lead to inadvertent errors.

Best practice

Avoid using type inference for function return types.

Interested readers may wonder why using `auto` when initializing variables is okay, but not recommended for function return types. A good rule of thumb is that `auto` is okay to use when defining a variable, because the object the variable is inferring a type from is visible on the right side of the statement. However, with functions, that is not the case -- there's no context to help indicate what type the function returns. A user would actually have to dig into the function body itself to determine what type the function returned. It's much less intuitive, and therefore more error prone.

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 declaration:

```
int add(int x, int y);
```

Using `auto`, this could be equivalently written as:

```
auto add(int x, int y) -> int;
```

In this case, `auto` does not perform type inference -- 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;
```

For now, we recommend the continued use of the traditional function return syntax. But we'll see this trailing return type syntax crop up again in lesson **7.15 -- Introduction to lambdas (anonymous functions)**.

Type inference for function parameter types

Many new programmers try something like this:

```
#include <iostream>

void addAndPrint(auto x, auto y) // only valid starting in C++20
{
    std::cout << x + y;
}

int main()
{
    addAndPrint(2, 3); // int
    addAndPrint(4.5, 6.7); // double
}
```

Prior to C++20, this won't work, because the compiler can't infer types for function parameters x and y at compile time. Pre-C++20, if you're looking to create generic functions that work with a variety of different types, you should be using `function templates` (covered in a later chapter), not `type inference`.

Starting in C++20, the `auto` keyword can be used as a shorthand way to create `function templates`, so the above code will compile and run. Note that this use of `auto` does not perform type inference.

For advanced readers

`Lambda expressions` have supported `auto` parameters since C++14. We'll cover lambda expressions in a future lesson.

As an aside...

If you're wondering what the `auto` keyword did pre-C++11, it was used to explicitly specify that a variable should have `automatic duration`:

```
int main()
{
    auto int foo(5); // explicitly specify that foo should have automatic duration

    return 0;
}
```

However, since all local variables in modern C++ default to `automatic duration` unless otherwise specified, the `auto` keyword was superfluous, and thus obsolete.

6.15 -- Implicit type conversion (coercion)

Index

6.13 -- Typedefs and type aliases

C++ TUTORIAL | PRINT THIS POST

58 comments to 6.14 — The auto keyword

Henry
October 27, 2019 at 10:21 am · Reply

So basically, c++ keyword auto has the same purpose as c#, javaScripts etc keyword var?

nascardriver
October 28, 2019 at 3:37 am · Reply

Yes

Fernando
September 4, 2019 at 2:09 pm · Reply

Hello!

1) Is there any performance-wise drawback for using "auto"?

```
1 // When performance is critical, should I use this:
2 double x = 5.3;
3 // instead of:
4 auto x = 5.3;
```

2) You mentioned that the compiler might guess your variable type wrong (not using these words though, and correct me if I'm wrong). Does this mean that duck-typed programming languages like Python suffer from this drawback by design?

Thanks

Alex
September 4, 2019 at 4:39 pm · Reply

1) No.
2) The compiler won't guess it wrong. However, if you use auto as a function return value,

the caller of the function might get it wrong. This could cause any number of problems to manifest, possibly resulting in incorrect or undefined behavior.

Edwin Martens
August 26, 2019 at 5:09 am · Reply

auto is evil !
why ?

auto value = doSomethingWith(int x);

The compiler may see the type of value, BUT YOU DO NOT !!!

what type is value here ?, anyone ?

Alex
August 27, 2019 at 1:18 pm · Reply

Auto isn't evil, it just needs to be used appropriately.

Your example would be a misuse, as it gives no context as to what type of return value to expect. But something like:

```
1 auto dictionary = getDictionary();
```

is fine -- it's clear we're returning some kind of Dictionary object.

A good rule of thumb is: "Use auto when it's hard to write the type, but the type is obvious".

Anthony
August 3, 2019 at 5:49 am · Reply

I've been using the auto keyword quite freely. But I can't figure out what the actual type is in this example:

```
1  template <class T, class A = std::allocator<T>>
2  class container {
3  public:
4      ...
5      container(size_type n, const alloc_type &a = alloc_type()) : ... {}
6      class iterator; // forward declaration for iterator
7      ...
8      iterator begin() { return iterator(this, 0); }
9      iterator end() { return iterator(this, size()); }
10
11     class iterator : public std::iterator<std::random_access_iterator_tag, value_type, diff
12     public:
13         ...
14     private:
15         ...
16     };
17
18 private
19     ...
20 };
21
22 int main() {
23     container<char> cont{8};
24
```

```
25      auto it1 = cont.begin();  // works fine, but what type is being returned?
26      container<char>::iterator it2 = cont.begin();  // compiler error: expected primary-expr
27      container<char, std::allocator<char>>::iterator it3 = cont.begin();  // compiler error:
28
29      return 0;
30  }
```

nascardriver
August 3, 2019 at 5:57 am · Reply

```
1  container<char>::iterator
```

I can't reproduce your error. Please post a minimal example that causes the error.

Anthony
August 3, 2019 at 6:34 am · Reply

Not sure what you mean. Should I post a minimal compilable version?

Btw, when I hover over 'begin' in both MSVC AND VSCode, it says 'container<char>::iterator' and I'm very surprised this isn't correct.

nascardriver
August 3, 2019 at 7:07 am · Reply

It is `container<char>::iterator`. There's something else wrong in your code.

> Should I post a minimal compilable version?
"compilable" is going to be hard, because your code doesn't compile. Post a minimal version that causes your error and no other errors.

Anthony
September 11, 2019 at 9:10 am · Reply

Hi,

Finally returned to this problem. As previously stated, @auto works, but when I attempt to do it 'manually', I get an error. Here's code showing the problem:

```
1   #include <iterator>
2
3   template <class T, class A = std::allocator<T>>
4   class ring {
5
6   public:
7       typedef T                                   value_type;  // yes,
8       typedef A                                   alloc_type;
9       typedef ring<value_type, alloc_type>        self_type;
10      typedef ptrdiff_t                           size_type;
11      typedef typename alloc_type::difference_type    difference_type;
12      typedef typename alloc_type::pointer            pointer;
13      typedef typename alloc_type::reference          reference;
14      typedef typename alloc_type::const_pointer      const_pointer;
15      typedef typename alloc_type::const_reference    const_reference;
16      class iterator;  // subclass, therefore no need for typedef
```

```
17      class const_iterator;
18
19      size_type size() { return 0; }  // just a stub for now
20
21      iterator begin() { return iterator(this, 0); }
22      iterator end() { return iterator(this, size()); }
23
24      class iterator : public std::iterator<std::random_access_iterator_tag
25      private:
26          self_type *ring_;  // typedefs defined in ring class apply here
27          size_type offset_;
28      public:
29          iterator(self_type *r, size_type o) : ring_{r}, offset_{o} {}
30      };
31
32  private:
33      size_type cap_;  // the capacity of the array
34      std::uint_fast8_t mask_;  // bitmask for options
35      alloc_type alloc_;  // our allocator
36      pointer array_;  // our base pointer for memory allocated by our allo
37      size_type head_;  // read/consume/front
38      size_type tail_;  // write/produce/back
39      bool empty_;  // true if empty
40  };
41
42  int main() {
43
44      ring<char> ring;  //  I have stripped out the actual constructor for
45      auto foo1{ring.begin()};  // this works
46      // ring<char>::iterator foo2 = ring.begin();  // this will cause the
47
48      return 0;
49  }
```

nascardriver
September 11, 2019 at 11:36 pm · Reply

You have one "ring" in line 4 and another "ring" in line 44. Line 45 only sees the "ring" from line 44. Use different name styles for variables, functions and types.

Anthony
September 13, 2019 at 11:36 am · Reply

I will NEVER make this mistake again :)
Thanks.

CHERIS
July 9, 2019 at 4:46 am · Reply

When will you start advance c++ topics. I really need it.

Alex
July 9, 2019 at 11:07 am · Reply

You can look at the front page of the site to see what topics are covered when.

Jeroen P. Broks
February 17, 2019 at 2:44 pm · Reply

Perhaps I'm a bit early, but I am really wondering if auto makes my life easier.

```
1 auto a = 1;
2 auto b = 1.5;
3 // in stead of
4 int a = 1;
5 float b = 1.5
6 // didn't save me a lot of typing, if you ask me...
```

I do see the sense in Go where auto is not needed

```
1 var a int = 1
2 var a = 1 // same as line above, as 1 is int that type is given, so that saves me some typin
3 a:=1 // declares a and assigns 1 and since 1 is an int, that type is given. Only works on lo
```

I could see the sense if something like this were possible:

```
1 auto{
2     a{1};
3     b{1.5};
4     c{10L};
5 }
```

Well, I guess not... but maybe I am just getting ahead of things.

(BTW. Just to note... This is the 6th C++ tutorial I've seen, but the first one in which a simple "Hello World" example didn't give me compile errors, already, and this tutorial made me write the first C++ code that actually WORKS. I always felt silly that I've been coding for almost 35 years and I could never master C and C++, so thanks a lot). ;)

nascardriver
February 18, 2019 at 3:54 am · Reply

Snippet 1
* Line 1, 2, 4: Initialize your variables with brace initializers. You used copy initialization.

```
1 auto b = 1.5;
2 // and
3 float c = 1.5;
4 // are not the same. @b is a double, as 1.5 is a double.
5 // If you want floats, use the 'f' postfix.
6 // If you used brace initialization, you would've noticed, because
7 float d{ 1.5 };
8 // doesn't compile.
```

@auto comes in handy when you get to templates. Types can fill up entire lines

```
1 // I'll have to use content from the future to get an example.
2 // For whatever reason, we want to have a vector (A list, not a mathematical
3 // vector) of strings where each character is an int.
4 // We have to specify the type once at the declaration.
5 std::vector<std::basic_string<int>> v{};
6
```

```
// Now we want an iterator (A pointer that knows how to get to the next element) to @
std::vector<std::basic_string<int>>::iterator it{ v.begin() };
// Great fun spelling that out all the time.
// Note that we could make the type infinitely longer.
// @auto to the rescue
auto it{ v.begin() };
```

Gurdeep Singh
December 10, 2018 at 3:33 pm · Reply

[auto calculateThis(int x, double d) -> std::string;] How it evauluates to std::string. Should'nt it be double?

nascardriver
December 11, 2018 at 8:16 am · Reply

Hi!

Without knowing what @calculateThis does, it could return anything. Judging by the function's name, you're right, it should most likely return an int or double.

Alex
December 11, 2018 at 12:36 pm · Reply

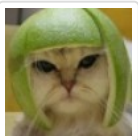
The trailing return type syntax moves the return type from before the function name to after the function name.

"auto calculateThis(int x, double d) -> std::string" clearly returns a std::string (as the "-> std::string" part of the declaration indicates).

auto in this context doesn't perform type inference in this context, it just acts as a placeholder.

nascardriver
December 12, 2018 at 3:22 am · Reply

Yep, that's right. There's nothing wrong with the code. It's just that me, and apparently Gurdeep, don't see why a function called "calculateThis" would return a string.



Alex
December 14, 2018 at 7:21 pm · Reply

Oh! I suppose that's true. I updated the function name and parameters to something that's more likely to be seen in the wild.

magaji::hussaini
November 18, 2018 at 11:36 am · Reply

I am not in love with this "auto" keyword to be used for variables either, makes me kinda lazy not to even care about my variable type just like dynamically typed languages cant even use unsigned types, and I think it increases compilation time also

Alex
November 18, 2018 at 3:14 pm · Reply

Wait until you get into nested template types. Then you'll change your mind. :)

magaji::hussaini
November 19, 2018 at 12:01 pm · Reply

Thanks...I am eager to

Pabitra Padhy
August 4, 2018 at 11:43 pm · Reply

Hey Alex,

Although Type Inference could be applied to members of struct or class
we have to make them static and const, then we could initialize them.

I don't see any use for this.
Could you care to explain, what could be the uses for such an use of Type Inference ?

Blaqsmite
June 12, 2018 at 11:32 pm · Reply

hi Alex
Am I doing something wrong or is ok to say that auto cant be used to define a variable in a .h file? I am attempting to do this in visual studio 2017 which i am guessing uses vc2015. I dont what version of Cpp it uses.
great tutorials by the way

nascardriver
June 13, 2018 at 1:19 am · Reply

Hi Balqsmite!

auto can be used in header files just fine, without code I'm guessing you're not initializing the variable in which case there's no way for your compiler to know which data type it should be.

himanshu shivnani
March 2, 2018 at 11:27 pm · Reply

The article is very useful Alex. Keep it up!!

vd
August 4, 2017 at 7:29 pm · Reply

Precise and well written. Thank you for this tutorial!

antiriad7
May 31, 2017 at 7:33 am · Reply

```
 void mySwap(auto &x, auto &y)
{
    auto z = x;
    x = y;
    y = z;
}
```

Why did u write &x and &y and not x and y in function parameters?

Alex
May 31, 2017 at 2:40 pm · Reply

My error. Those are reference parameters, which I haven't covered yet. I've updated the example to one that does not use reference parameters.

Lucas Aaron
October 3, 2017 at 6:50 am · Reply

When you are so smart you accidently use smart things in easy tutorials

Alex
October 3, 2017 at 3:16 pm · Reply

More to the point, it's hard to break habits and do things inefficiently because you haven't covered the optimal way to do it yet. :)

Peter
May 16, 2017 at 2:32 pm · Reply

> This is called automatic type deduction.

Actually AFAIK it's simply called type inference - https://en.wikipedia.org/wiki/Type_inference

Alex
May 17, 2017 at 10:40 am · Reply

Thanks for the heads up. I've updated the lesson accordingly!

Zachary Fojtasek
December 5, 2016 at 10:57 am · Reply

Why can I not do something like this?:

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

this does not compile.

But if it did, I could use a function like this instead of overloading functions.

Alex
December 5, 2016 at 1:56 pm · Reply

Auto tells the compiler to infer the proper type -- because the compiler does the inferring, the type has to be inferable at compile-time. Your example doesn't work because x and y (and the thus the return type) can't be determined at compile time.

However, C++ does provide functionality to do what you want. In chapter 13 I talk about function templates, which are designed to do exactly what you're intending here.

(Note that this restriction may be lifted in future versions of C++)

selami
November 11, 2016 at 9:23 am · Reply

auto keyword can easily be used by int, double, char etc.. but not used with string.
why? when used. it is written as PKc. what does it mean?

Alex
November 13, 2016 at 1:10 pm · Reply

Auto works just fine with std::string (and char*), so not sure what you mean when you say it can't be used with string.

Matt
November 1, 2016 at 2:01 pm · Reply

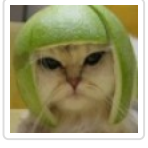
Can the auto keyword be used for function return type deduction when the return type is void(or as part of the trailing return type syntax)?

Alex
November 1, 2016 at 6:33 pm · Reply

Yes. I've updated to lesson to explicitly indicate so.

Matt
November 1, 2016 at 1:54 pm · Reply

In the first paragraph, the wording needs to be updated to say that local variables are created upon definition(not when entering the block).
Hope I'm not getting annoying :)



Alex
November 1, 2016 at 6:26 pm · Reply

Not annoying at all -- on the contrary, you're making the lessons better for all the future readers, and for that, I think you.

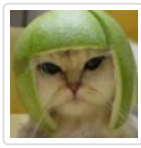
Dekasi

September 9, 2016 at 2:06 pm · Reply

"While this may seem neat, we recommend that this syntax be avoided for functions that return a fixed type. The return type of a function is of great use in helping to document a function. When a specific type isn't specified, the user may not know what is expected.

(Side note: Interested readers may wonder why using auto when initializing variables is okay, but not recommended for function return types. A good rule of thumb is that auto is okay to use within the boundaries of a statement, because the type being inferred often doesn't matter, and if it does, the actual type information is generally at hand. However, across statement boundaries, it's better for documentation and maintenance to make types explicit.)"

I don't understand few things in this excerpt:
1. what are fixed types
2. to document a function
3. in which case you think that the user needs to know to type
4. what can be referred as a boundary of statement
5. `the type being inferred often doesn't matter, and if it does, the actual type information is generally at hand`- what you mean with this



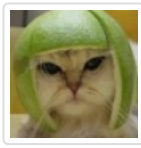
Alex
September 9, 2016 at 2:51 pm · Reply

Rather than answer all of these questions, I updated the wording of that section substantially. Have another read and see if it's clearer now. If not, let me know what's still confusing.

shreyanshd
May 25, 2016 at 2:52 am · Reply

Would the use of automatic type deduction in C++ affect the performance? If auto keyword is used extensively in C++ then the compiler has to figure out the data type every time, which would degrade performance.

Correct me if I am wrong.
Great tutorials. Love them :)



Alex
May 25, 2016 at 7:30 pm · Reply

No, the compiler already needs to know the type in order to do strong type checking (to give you an error if you try to do something nonsensical). Auto is really just a convenience for the programmer.

Ola Sh
May 4, 2016 at 12:03 pm · Reply

Thanks for the good work. In my opinion, this is the best programming tutorial that I have used. Thanks again.

Sandro
January 22, 2016 at 9:42 pm · Reply

"auto d = 5.0; // 5.0 is a double literal, so d will be type double"

It's a double literal because there is no "f" or "l" at the end of the number?

Alex
January 23, 2016 at 9:23 am · Reply

Correct.

hridayesh
November 1, 2015 at 8:45 am · Reply

```
#include<iostream>

int main()
{

        using namespace std;

        int i = 23;
        float j = 234.34;
        auto k = i+j;
        auto l = 12+34;
        cout<<l<<endl;
        cout<<k;
        return 0;

}
```

hey alex i am using codeblocks and this program is throwing error saying that k and l are not declared.

Alex
November 5, 2015 at 9:31 pm · Reply

It sounds like your code::blocks isn't set to use C++11 functionality. See **http://stackoverflow.com/questions/18174988/how-can-i-add-c11-support-to-codeblocks-compiler** for a solution.

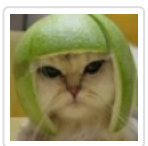
programmer, another one
August 19, 2015 at 8:16 am · Reply

hello alex,

I used the auto keyword in a class as a public member on two variables in Qt creator, and the compiler flagged it as an error saying that: "a non-static data member cannot have a type that contains auto".

do you know why this may have occurred?

thanks



Alex
August 19, 2015 at 1:50 pm · Reply

It looks like struct and class members can only be made auto if they are also static.

Avneet
July 31, 2015 at 8:16 am · Reply

Yup Boyan is right. Remove "we will" from the following sentence:
"While using auto in place of fundamental data types only saves a few (if any) keystrokes, in future lessons we'll we will see examples where the types get complex and lengthy."
Just highlighting the typo made in this section. :)

Boyan
July 30, 2015 at 9:18 pm · Reply

Just above the "Automatic type deduction for functions in C++14" headline, you've written "we'll we will see "

I've got to say though...this is probably the best tutorial I've seen on ANYTHING, let alone C++! Thank you for the awesome work! :)

Alex
July 31, 2015 at 8:46 am · Reply

Typo fixed. Thanks for the compliment.