6.10 — Static local variables

BY ALEX ON JUNE 19TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 3RD, 2020

The term `static` is one of the most confusing terms in the C++ language, in large part because `static` has different meanings in different contexts.

In prior lessons, we covered that global variables have `static duration`, which means they are created when the program starts and destroyed when the program ends.

We also discussed how the `static` keyword gives a global identifier `internal linkage`, which means the identifier can only be used in the file in which it is defined.

In this lesson, we'll explore the use of the `static` keyword when applied to a local variable.

Static local variables

In lesson **2.4 -- Introduction to local scope**, you learned that local variable have `automatic duration` by default, which means they are created at the point of definition, and destroyed when the block is exited.

Using the `static` keyword on a local variable changes its duration from `automatic duration` to `static duration`. This means the variable is now created at the start of the program, and destroyed at the end of the program (just like a global variable). As a result, the static variable will retain its value even after it goes out of scope!

The easiest way to show the difference between `automatic duration` and `static duration` variables is by example.

Automatic duration (default):

```
#include <iostream>

void incrementAndPrint()
{
    int value{ 1 }; // automatic duration by default
    ++value;
    std::cout << value << '\n';
} // value is destroyed here

int main()
{
    incrementAndPrint();
    incrementAndPrint();
    incrementAndPrint();

    return 0;
}
```

Each time incrementAndPrint() is called, a variable named value is created and assigned the value of 1. incrementAndPrint() increments value to 2, and then prints the value of 2. When incrementAndPrint() is finished running, the variable goes out of scope and is destroyed. Consequently, this program outputs:

```
2
2
2
```

Now consider the static version of this program. The only difference between this and the above program is that we've changed the local variable from `automatic duration` to `static duration` by using the `static` keyword.

Static duration (using static keyword):

```
#include <iostream>

void incrementAndPrint()
{
    static int s_value{ 1 }; // static duration via static keyword.  This initializer is only executed once.
    ++s_value;
    std::cout << s_value << '\n';
} // s_value is not destroyed here, but becomes inaccessible because it goes out of scope

int main()
{
    incrementAndPrint();
    incrementAndPrint();
    incrementAndPrint();

    return 0;
}
```

In this program, because `s_value` has been declared as `static`, `s_value` is created (and zero-initialized) once (at program start). The variable is then initialized with our supplied initialization value (1) the first time the variable definition is encountered (but it is not reinitialized on subsequent calls).

When `s_value` goes out of scope at the end of the function, it is not destroyed. Each time the function incrementAndPrint() is called, the value of `s_value` remains at whatever we left it at previously. Consequently, this program outputs:

```
2
3
4
```

Just like we use "g_" to prefix global variables, it's common to use "s_" to prefix static (static duration) local variables.

One of the most common uses for static duration local variables is for unique ID generators. Imagine a program where you have many similar objects (e.g. a game where you're being attacked by many zombies, or a simulation where you're displaying many triangles). If you notice a defect, it can near impossible to distinguish which object is having problems. However, if each object is given a unique identifier upon creation, then it can be easier to differentiate the objects for further debugging.

Generating a unique ID number is very easy to do with a static duration local variable:

```
int generateID()
{
    static int s_itemID{ 0 };
    return s_itemID++; // makes copy of s_itemID, increments the real s_itemID, then returns the value in the copy
}
```

The first time this function is called, it returns 0. The second time, it returns 1. Each time it is called, it returns a number one higher than the previous time it was called. You can assign these numbers as unique IDs for your objects. Because `s_itemID` is a local variable, it can not be "tampered with" by other functions.

Static variables offer some of the benefit of global variables (they don't get destroyed until the end of the program) while limiting their visibility to block scope. This makes them safe for use even if you change their values regularly.

Quiz time

Question #1

What effect does using keyword `static` have on a global variable? What effect does it have on a local variable?

Show Solution

6.11 -- Scope, duration, and linkage summary

Index

6.9 -- Why global variables are evil

C++ TUTORIAL | PRINT THIS POST

112 comments to 6.10 — Static local variables

« Older Comments 1 2

Tom Bauer
February 4, 2020 at 9:35 am · Reply

So, if you call a funktion that creates a static variable again and again it doesnt matter that the variable is created over and over again? Or is it just created once in global scope and the other "creations" are ignored?

AbraxasKnister
February 6, 2020 at 6:11 am · Reply

Freely interpreted from what the text after the second code snippet says about the lifecycle of a static local variable: (a) It gets created and initialized to 0 at program startup (b) its value is overridden by the value given at the variable definition at first encounter (ie. at the time the respective function is called the first time) (c) in subsequent calls its definition statement is effectively ignored (d) at termination of the function the variables value is stored (e) subsequent function calls function as if the variable was defined with precisely the value that was stored for it the last termination. TL;DR yes. (to 2nd question)

Tom Bauer
February 7, 2020 at 7:37 am · Reply

Thanks

koe
December 21, 2019 at 9:17 pm · Reply

In the next section it mentions that static local variables are instantiated at program startup. Is this accurate, and if so does it mean all static local variables will be created regardless of if their block is actually executed?

nascardriver
December 22, 2019 at 4:29 am · Reply

`static` variables are created at program startup. Creation means memory reservation, but not initialization. Local `static` variables are initialized when they are first encountered during execution, but their memory already existed. Global `static` variables are initialized at program-startup.
If you declare a local `static` variable but execution doesn't reach it, you still used the memory (But never initialized it).

Arush Agarampur
December 16, 2019 at 8:05 pm · Reply

When I make a class like

```
1 Class C
2 {
3     static int IntVal;
4 public:
5     auto OutputPrivateValue() -> void { return IntVal; }
6 };
```

I must do

```
1 int C::IntVal = 0;
```

to initialize and access it. I understand how to do it, but why do I have to do it? And what is this called?

nascardriver
December 17, 2019 at 2:22 am · Reply

Line 3 is only a declaration of `IntVal`, not a definition. You can't define it in the header, because every file that includes the header would defined `IntVal`. That's a violation of the one-definition-rule.
`int C::IntVal{ 0 };` is the definition of `IntVal`. It goes in the source file that corresponds to the header `C` was declared in. The source file only gets compiled once, so you don't get multiple definitions.

NooneAtAll
December 10, 2019 at 6:33 am · Reply

Patiently waiting for addition of constinit to this page...

nascardriver
December 11, 2019 at 4:59 am · Reply

`constinit` will probably be added alongside `constexpr` functions.
`constexpr` functions will not be covered before C++20 is available in the major compilers, because C++20 lifted many restrictions on `constexpr` functions.

I haven't used `constinit` yet, but as far as I can tell it doesn't change a whole lot. You should be able to understand it from reading its documentation.

NooneAtAll
December 16, 2019 at 1:59 pm · Reply

ha, you fell for this just like me

functions can be const**eval**
const**init** is to make static variables be calculated at compile time, instead of first call of function

Samira Ferdi
September 18, 2019 at 5:34 pm · Reply

Hi, Alex and Nascardriver!

Does right if constexpr/const is actually static(by default) global variable?

nascardriver
September 19, 2019 at 12:32 am · Reply

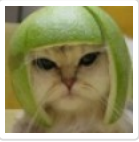
`constexpr` variables are `const`. `const` variables have internal linkage by default.

Samira Ferdi
September 19, 2019 at 4:50 pm · Reply

Thank you, Nascardriver!

Anonymous
March 5, 2019 at 2:06 am · Reply

Why are global variables evil while static aren't?



Alex
March 8, 2019 at 12:47 pm · Reply

Static local variables are scoped to the local functions in which they are declared.

It's not the persistence that's evil, it's the broad accessibility by anyone that's evil.

error
February 24, 2019 at 4:03 pm · Reply

I think there is a typo here (2nd paragraph to the last): " Because s_itemID is a local variable, it can not be “tampered with” by other functions."

s_itemID is a static variable, not a local variable...is it not?

Kodnot
February 26, 2019 at 1:40 am · Reply

Nope, not a typo. s_itemID is a static local variable - static duration, local scope. And it cannot be "tampered with" by other functions due to its local scope, while retaining its value between function calls due to its static duration.

Jeroen P. Broks
February 17, 2019 at 11:07 am · Reply

Now the "static" keyword the way it's explained here was something that really surprised me when I was taking courses like these about C. C was the first language of which I found out they are supported (although since C and C++ are the only language for which I am actually reading these very detailed tutorials I may have missed stuff in other languages), so I guess I may have used a lot of globals in the past which would better have served as 'static' variables.

Now I began coding when I was 8 or 9 or so, and now I'm in my forties, and old habits die hard, so I guess, I might be guilty of the crime for using unneeded globals awhile longer :P

Puya
January 19, 2019 at 5:14 pm · Reply

Hi Alex,

I think this might be a typo "automatic duration [means] they are created when the block is entered", since in 4.1a it says: "automatic duration [means] they are [created] at the point of definition". (which would be different if your definition is not at the top of your block)
e.g.

```
1 {
2     std::cout << name << std::endl; // might lead to error
3     std::string name = "Sue";
4 }
```

Alex
January 23, 2019 at 8:53 pm · Reply

It's more correct to say they're created at the point of definition. I've updated this article accordingly. Thanks for pointing out the mismatch.

Benur21
January 18, 2019 at 11:56 am · Reply

1)

```
1 static int s_value = 0;
2 static int s_value = 1;
```

If I do like this it will only execute the first (0)?

2)
When doing

```
1 static int s_value = 1;
```

, if a previous static int s_itemID is accessible (in the right scope) it does not execute; and if s_itemID is not accessible (in some other scope or not created at all) it will execute?

3)
Does "static int s_value" conflict with other previously created variables in outside blocks, as well as with variables of different types with same name?

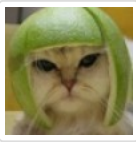
nascardriver
January 19, 2019 at 8:05 am · Reply

These are all questions you can find out yourself by trying. That way you'll think about the code and the chances that you remember the answers are higher.

Codename 47
December 17, 2018 at 1:34 am · Reply

Hey Alex,

Only for the sake of consistency -- in the first two examples in this tutorial, the function int main() does not have any return statement.



Alex
December 17, 2018 at 1:49 pm · Reply

Consistentized. Thanks for pointing this out!

diksha chadha
December 5, 2018 at 10:14 pm · Reply

Hi,

I have a doubt in the line :
Static variables offer some of the benefit of global variables (they don’t get destroyed until the end of the program) "while limiting their visibility to block scope". This makes them much safer for use than global variables.

And in 8.11 it is said : Note that this static member definition is not subject to access controls: you can define and initialize the value even if it’s declared as private (or protected) in the class.

Could you please explain the difference in the two.

Alex
December 6, 2018 at 12:27 pm · Reply

Two different things entirely. The static variables in this lesson are for non-members, the ones in 8.11 are for static members.

Non-member static variables are typically defined in a function. They retain their value across function calls, but can't be accessed outside the function.
Member static variables are defined inside a class. Although normal private or protected members can't

be accessed directly from outside the class, a static member definition & initialization can be (but not access beyond that).

I'm in love with alex's tutor
July 23, 2018 at 3:03 am · Reply

Dear Great Teacher Mr. Alex and great supporter Mr/Mrs. nascardriver

1) I tried to execute the above program in tutor 4.3 that have a function named incrementAndPrint(); with both static int s_value = 1;(declared as a local variable) and static int g_value = 1;(declared as a global variable) and i recognized the program executes the same output.
Depend on this fact are we able to conclude that the effect of static key word is the same in both local and global(Internal linkage) variable declaration? If we can't what are the reasons?
2) This program has also a variable s_value with static duration and eventually outputs 2,3 and 4. Can you explain in more detail how it works? I have tried my best to know about it even by greeting back to read the previous sections.B/s i have seen that you are so busy with a lot of questions coming from others.
when the 1st incrementAndPrint(); function called it increment the value 1 to 2.what about the out put 3 (2nd function output) Does that mean the value of the first function would copied the value 2 to the secondly called function?
Did we cover this idea in the previous chapter? where?
God bless you all!!! You are changing my life indeed!!

nascardriver
July 23, 2018 at 3:46 am · Reply

1) Static variables behave just like global variables, they're unique to the program.
2)

```
#include <iostream>

// Based on question 1, imagine it like this.
int s_value{ 1 };

void incrementAndPrint()
{
  ++s_value;
  std::cout << s_value << '\n';
}

int main()
{
  incrementAndPrint();
  incrementAndPrint();
  incrementAndPrint();

  return 0;
}
```

@s_value is only initialized one and maintains it's value across function calls.

I'm in love with alex's tutor
July 23, 2018 at 6:01 am · Reply

Dear Mr/Mrs.nascardriver
Thank you so much for your kind,prompt and clear response . Now i have gotten the point for static variables that they behave just like global variables except they have no linkage like

global variables!!!
And i like the way you clarify my number 2 question.

I wish God be always with you all !!!
Thank you indeed!

Nux
April 28, 2018 at 1:43 am · Reply

Hello,

I remember you was willing to change std::endl to "\n" in exemple

The 2 first exemple here use std::endl

Alex
April 29, 2018 at 11:28 am · Reply

Thanks! I've updated the examples.

Hema
March 10, 2018 at 10:16 pm · Reply

Hello, I was experimenting with the calling functons. I wrote the following code. The compiler says: error:expected primary-expression before 'int'. The error is in line 11.
Precisely here --> cout<<"x+y= "<< int x()+ int y()<<endl;
^

```
1  int x(int x){
2      x=8;
3      return x;
4  }
5  int y(int y){
6      y=9;
7      return y;
8  }
9
10 int main(){
11     cout<<"x+y= "<< int x()+ int y()<<endl;
12     return 0;
13 }
```

What does the compiler actually mean and how can I correct it?
Thanks..

nascardriver
March 11, 2018 at 6:45 am · Reply

Hi Hema!

First, let's rename some variables to prevent duplicate names

```
1  #include <iostream>
2
3  // Don't use 'using namespace'
4
5  int x(int i)
6  {
```

```
  i = 8;
  return i;
}

int y(int i)
{
  i = 9;
  return i;
}

int main()
{
  std::cout << "x+y= " << int x() + int y() << std::endl;

  return 0;
}
```

The error is still the same, let's look into it.
You only need to specify the return and parameter types of a function when declaring it. Not when calling it. So when you want to call @x you don't write int.

```
#include <iostream>

int x(int i)
{
  i = 8;
  return i;
}

int y(int i)
{
  i = 9;
  return i;
}

int main()
{
  // Removed 'int'
  std::cout << "x+y= " << x() + y() << std::endl;

  return 0;
}
```

Now you're getting another error:

```
error: too few arguments to function ‘int x(int)’
   std::cout << "x+y= " << x() + y() << std::endl;
                              ^
```

When you declared @x you decided it should get a parameter (@i). When you call a function the requires parameters you need to pass them in the brackets.

```
#include <iostream>

int x(int i)
{
  i = 8;
  return i;
}

int y(int i)
{
  i = 9;
```

```
12    return i;
13  }
14
15  int main()
16  {
17    // Added 123 and 456
18    std::cout << "x+y= " << x(123) + y(456) << std::endl;
19
20    return 0;
21  }
```

Your code now compiles and outputs

```
1  x+y= 17
```

Keep on experimenting, it's a good way to learn!

Hema
March 12, 2018 at 9:28 pm · Reply

Can I know why we shouldn't use "using namespace"? The program compiles fine with "using namespace" after removing "int" in line 11.

Thanks a lot for the help. :)

nascardriver
March 13, 2018 at 2:45 am · Reply

'using namespace' can lead to naming conflicts. eg. The std namespace has a function called 'min'. This is quite a common name that could occur somewhere else too. You you were 'using namespace std' and called 'min' your compiler wouldn't be able to tell which 'min' you're trying to call.

Reference
Lesson 1.8a Naming conflicts and the std namespace

Alex
March 13, 2018 at 3:16 pm · Reply

Also discussed more in lesson 4.3c.

Trong Nguyen Van
March 3, 2018 at 10:44 pm · Reply

I see this is very nice. This static variable isn't destroyed after exiting its scope but we can only access to it in its scope again. In main() or other scope, we can't access it's value!

Thank you!

Hema
October 17, 2017 at 3:42 am · Reply

Can you eplain this sentence in more detail-"Static duration variables are only created (and initialized) once, and then they are persisted throughout the life of the program."

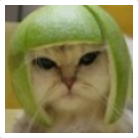
Thanks

Alex
October 21, 2017 at 9:58 am · Reply

Sure. Normally, if you have a variable defined inside a function (either as a parameter or local variable), that variable is created and destroyed every time the function is executed. There's no persistence. However, with a static variable, the static variable is only created the first time the function is run, and it is not destroyed when the function exits. The next time the function is run, the variable still exists and its previous value has been retained. The static variable is only destroyed at the end of the entire program.

felipe
March 17, 2017 at 6:13 pm · Reply

Is there a super static version of static(that makes the variable value persist even after the program is close?) thanks.

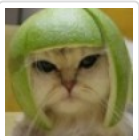


Alex
March 20, 2017 at 8:08 am · Reply

Nope. If you want to persist a value across multiple program executions, you'll need to write it to disk and then read it back in.

Kanwar Ujjwaldeep Singh
January 14, 2017 at 4:03 am · Reply

VoidIncrementandPrint
This Function has no return value,how it send's back value 2 bact to MAIN FUNCTION.
When main function calls VoidIncrementandPrint, Value is created ,assigned value =1,than incremented and then destroyed at end.But its not sent back to the main Function.Than how it works>



Alex
January 16, 2017 at 7:37 pm · Reply

It doesn't send any value back to main(). It prints the value itself.

« Older Comments 1 2