

15.6 — Static member variables

👤 **ALEX¹** 🕒 **DECEMBER 2, 2024**

In the lesson [7.4 -- Introduction to global variables](https://www.learncpp.com/cpp-tutorial/introduction-to-global-variables/)², we introduced global variables, and in lesson [7.11 -- Static local variables](https://www.learncpp.com/cpp-tutorial/static-local-variables/)³, we introduced static local variables. Both of these types of variables have static duration, meaning they are created at the start of the program, and destroyed at the end of the program. Such variables keep their values even if they go out of scope.

For example:

```
1 | #include <iostream>
2 |
3 | int generateID()
4 | {
5 |     static int s_id{ 0 }; // static local variable
6 |     return ++s_id;
7 | }
8 |
9 | int main()
10 | {
11 |     std::cout << generateID() << '\n';
12 |     std::cout << generateID() << '\n';
13 |     std::cout << generateID() << '\n';
14 |
15 |     return 0;
16 | }
```

This program prints:

```
1
2
3
```

Note that static local variable `s_id` has kept its value across multiple function calls.

Class types bring two more uses for the `static` keyword: static member variables, and static member functions. Fortunately, these uses are fairly straightforward. We'll talk about static member variables in this lesson, and static member functions in the next.

Static member variables

Before we go into the static keyword as applied to member variables, first consider the following class:

```

1  #include <iostream>
2
3  struct Something
4  {
5      int value{ 1 };
6  };
7
8  int main()
9  {
10     Something first{};
11     Something second{};
12
13     first.value = 2;
14
15     std::cout << first.value << '\n';
16     std::cout << second.value << '\n';
17
18     return 0;
19 }

```

When we instantiate a class object, each object gets its own copy of all normal member variables. In this case, because we have declared two `Something` class objects, we end up with two copies of `value`: `first.value`, and `second.value`. `first.value` is distinct from `second.value`. Consequently, the program above prints:

```

2
1

```

Member variables of a class can be made static by using the `static` keyword. Unlike normal member variables, **static member variables** are shared by all objects of the class. Consider the following program, similar to the above:

```

1  #include <iostream>
2
3  struct Something
4  {
5      static int s_value; // declare s_value as static (initializer moved below)
6  };
7
8  int Something::s_value{ 1 }; // define and initialize s_value to 1 (we'll discuss this
9  // section below)
10
11 int main()
12 {
13     Something first{};
14     Something second{};
15
16     first.s_value = 2;
17
18     std::cout << first.s_value << '\n';
19     std::cout << second.s_value << '\n';
20     return 0;
21 }

```

This program produces the following output:

2

2

Because `s_value` is a static member variable, `s_value` is shared between all objects of the class. Consequently, `first.s_value` is the same variable as `second.s_value`. The above program shows that the value we set using `first` can be accessed using `second`!

Static members are not associated with class objects

Although you can access static members through objects of the class (as shown with `first.s_value` and `second.s_value` in the example above), static members exist even if no objects of the class have been instantiated! This makes sense: they are created at the start of the program and destroyed at the end of the program, so their lifetime is not bound to a class object like a normal member.

Essentially, static members are global variables that live inside the scope region of the class. There is very little difference between a static member of a class and a normal variable inside a namespace.

Key insight

Static members are global variables that live inside the scope region of the class.

Because static member `s_value` exists independently of any class objects, it can be accessed directly using the class name and the scope resolution operator (in this case, `Something::s_value`):

```
1 | class Something
2 | {
3 | public:
4 |     static int s_value; // declare s_value as static
5 | };
6 |
7 | int Something::s_value{ 1 }; // define and initialize s_value to 1 (we'll discuss this
8 | section below)
9 |
10 | int main()
11 | {
12 |     // note: we're not instantiating any objects of type Something
13 |
14 |     Something::s_value = 2;
15 |     std::cout << Something::s_value << '\n';
16 |     return 0;
17 | }
```

In the above snippet, `s_value` is referenced by class name `Something` rather than through an object. Note that we have not even instantiated an object of type `Something`, but we are still able to access and use `Something::s_value`. This is the preferred method for accessing static members.

Best practice

Access static members using the class name and the scope resolution operator (`::`).

Defining and initializing static member variables

When we declare a static member variable inside a class type, we're telling the compiler about the existence of a static member variable, but not actually defining it (much like a forward declaration). Because static member variables are essentially global variables, you must explicitly define (and optionally initialize) the static member outside of the class, in the global scope.

In the example above, we do so via this line:

```
1 | int Something::s_value{ 1 }; // define and initialize s_value to 1
```

This line serves two purposes: it instantiates the static member variable (just like a global variable), and initializes it. In this case, we're providing the initialization value `1`. If no initializer is provided, static member variables are zero-initialized by default.

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 (as definitions are not considered to be a form of access).

For non-template classes, if the class is defined in a header (.h) file, the static member definition is usually placed in the associated code file for the class (e.g. `Something.cpp`). Alternatively, the member can also be defined as `inline` and placed below the class definition in the header (this is useful for header-only libraries). If the class is defined in a source (.cpp) file, the static member definition is usually placed directly underneath the class. Do not put the static member definition in a header file (much like a global variable, if that header file gets included more than once, you'll end up with multiple definitions, which will cause a linker error).

For template classes, the (templated) static member definition is typically placed directly underneath the template class definition in the header file (this doesn't violate the ODR because such definitions are implicitly inline).

Initialization of static member variables inside the class definition

There are a few shortcuts to the above. First, when the static member is a constant integral type (which includes `char` and `bool`) or a const enum, the static member can be initialized inside the class definition:

```
1 | class Whatever
2 | {
3 | public:
4 |     static const int s_value{ 4 }; // a static const int can be defined and
   |     initialized directly
5 | };
```

In the above example, because the static member variable is a const int, no explicit definition line is needed. This shortcut is allowed because these specific const types are compile-time constants.

In lesson [7.10 -- Sharing global constants across multiple files \(using inline variables\)](https://www.learncpp.com/cpp-tutorial/sharing-global-constants-across-multiple-files-using-inline-variables/)⁴, we introduced inline variables, which are variables that are allowed to have multiple definitions. C++17 allows static members to be inline variables:

```
1 | class Whatever
2 | {
3 | public:
4 |     static inline int s_value{ 4 }; // a static inline variable can be defined and
5 |     initialized directly
6 | };
```

Such variables can be initialized inside the class definition regardless of whether they are constant or not. This is the preferred method of defining and initializing static members.

Because `constexpr` members are implicitly inline (as of C++17), static `constexpr` members can also be initialized inside the class definition without explicit use of the `inline` keyword:

```
1 | #include <string_view>
2 |
3 | class Whatever
4 | {
5 | public:
6 |     static constexpr double s_value{ 2.2 }; // ok
7 |     static constexpr std::string_view s_view{ "Hello" }; // this even works for
8 |     classes that support constexpr initialization
9 | };
```

Best practice

Make your static members `inline` or `constexpr` so they can be initialized inside the class definition.

An example of static member variables

Why use static variables inside classes? One use is to assign a unique ID to every instance of the class. Here's an example:

```

1  #include <iostream>
2
3  class Something
4  {
5  private:
6      static inline int s_idGenerator { 1 };
7      int m_id {};
8
9  public:
10     // grab the next value from the id generator
11     Something() : m_id { s_idGenerator++ }
12     {
13     }
14
15     int getID() const { return m_id; }
16 };
17
18 int main()
19 {
20     Something first{};
21     Something second{};
22     Something third{};
23
24     std::cout << first.getID() << '\n';
25     std::cout << second.getID() << '\n';
26     std::cout << third.getID() << '\n';
27     return 0;
28 }

```

This program prints:

```

1
2
3

```

Because `s_idGenerator` is shared by all `Something` objects, when a new `Something` object is created, the constructor initializes `m_id` with the current value of `s_idGenerator` and then increments the value for the next object. This guarantees that each instantiated `Something` object receives a unique id (incremented in the order of creation).

Giving each object a unique ID can help when debugging, as it can be used to differentiate objects that otherwise have identical data. This is particularly true when working with arrays of data.

Static member variables are also useful when the class needs to utilize a lookup table (e.g. an array used to store a set of pre-calculated values). By making the lookup table static, only one copy exists for all objects, rather than making a copy for each object instantiated. This can save substantial amounts of memory.

Only static members may use type deduction (`auto` and CTAD)

A static member may use `auto` to deduce its type from the initializer, or Class Template Argument Deduction (CTAD) to deduce template type arguments from the initializer.


Non-static members may not use `auto` or CTAD.

The reasons for this distinction being made are quite complicated, but boil down to there being certain cases that can occur with non-static members that lead to ambiguity or non-intuitive results. This does not


occur for static members. Thus non-static members are restricted from using these features, whereas static members are not.

```
1 #include <utility> // for std::pair<T, U>
2
3 class Foo
4 {
5 private:
6     auto m_x { 5 };           // auto not allowed for non-static members
7     std::pair m_v { 1, 2.3 }; // CTAD not allowed for non-static members
8
9     static inline auto s_x { 5 };           // auto allowed for static members
10    static inline std::pair s_v { 1, 2.3 }; // CTAD allowed for static members
11
12 public:
13     Foo() {};
14 };
15
16 int main()
17 {
18     Foo foo{};
19
20     return 0;
21 }
```

5

 [Next lesson](#)
15.7 [Static member functions](#)


6

 [Back to table of contents](#)

7


 [Previous lesson](#)
15.5 [Class templates with member functions](#)

8






B U URL INLINE CODE C++ CODE BLOCK HELP!

Leave a comment...

 Name*

@ Email*

Notify me about replies: 

POST COMMENT

🔍 Find a mistake? Leave a comment above!?

👤 Avatars from <https://gravatar.com/>¹⁰ are connected to your provided email address.

305 COMMENTS

Newest ▼



Washington Humberto Peralta

🕒 April 28, 2025 4:31 pm PDT

April 28th, 2025

Good evening, Mister Alex,

Chapter 15.6

I was thinking in my previous question and I realize that the instruction number 11

Something() : m_id { s_idGenerator++ }

Is a constructor. Am I right?

Thank you for your help

Washington H. Peralta

👍 0 ➡ Reply



PollyWantsACracker

🗨 Reply to [Washington Humberto Peralta](#)¹¹ 🕒 May 15, 2025 8:03 am PDT

yes, btw you can modify your comments

✎ Last edited 1 month ago by PollyWantsACracker

👍 0 ➡ Reply



Washington Humberto Peralta

🕒 April 28, 2025 8:32 am PDT

April 28th , 2025

Good morning, Mister Alex,

Chapter 15.6

In the program that is below this text

“An example of static member variables

Why use static variables inside classes? One use is to assign a unique ID to every instance of the class.

Here’s an example:”

Is correct the instruction number 11

Something() : m_id { s_idGenerator++ }

Or the correct instruction should be

Something() :: m_id { s_idGenerator++ }

Thank you for your help

Washington H. Peralta

👍 0 ➡ Reply



Leni

🕒 March 9, 2025 9:37 pm PDT

Static member variables are shared across all instances of a class, enabling efficient memory use and facilitating features like unique ID generation. How can improperly managing static member variables lead to unintended side effects in multi-threaded applications?



0

↩ Reply



yiken

🕒 January 18, 2025 5:29 am PST

loading...

🔗 *Last edited 5 months ago by yiken*



1

↩ Reply



KLAP

🕒 January 16, 2025 1:52 pm PST

Is there any benefit to create a constructor for setting unique ID for objects like your example?

```
1 | class Something
2 | {
3 | private:
4 |     static inline int s_idGenerator { 1 };
5 |     int m_id {};
6 |
7 | public:
8 |     // grab the next value from the id generator
9 |     Something() : m_id { s_idGenerator++ }
10 |    {
11 |    }
```

Can I just do this instead and it produces the same result but I don't have to make a constructor for it?

```
1 | class Something
2 | {
3 | private:
4 |     static inline int s_idGenerator{ 1 };
5 |     int m_id{ s_idGenerator ++ };
6 |
7 | public:
8 |     int getID() const { return m_id; }
9 | };
```

🔗 *Last edited 5 months ago by KLAP*



0

↩ Reply



Initialization Timing of Inline Static vs Non-inli

🕒 January 10, 2025 10:57 am PST

```
1 | class Whatever
2 | {
3 | public:
4 |     static inline int s_value{ 4 }; // a static inline variable can be defined and
    initialized directly
5 | };
```

Such variables can be initialized inside the class definition regardless of whether they are constant or not. This is the preferred method of defining and initializing static members.

So Alex, are inline static class members instantiated in the class before main() starts? I thought they would be created when they are first accessed, similar to the First use idiom access. But it seems that for inline static class members, instantiation happens before main() starts, in the same way as non-inline static members. Am I right?

👍 0 ➡ Reply



NordicCat

👤 Reply to [Initialization Timing of Inline Static vs Non-inli](#) ¹² 🕒 January 13, 2025 4:50 am PST

broo are u serious u changed your name on the name of your doubt, XD: like fr are u okay bud??

👍 3 ➡ Reply



Albert

🕒 November 30, 2024 7:36 pm PST

One thing i dont get

```
1 | int Something::s_value{ 1 }; // defines the static member variable
```

Why exactly do we need that int type to initialized or define this static variable?

Typically when we have a declared variable in the global scope we can define in main() without needing to explicitly write the type.

//Global scope

Static int x;

//Main()

x{5}; // no need for int x{5};

Is it because its inside of a class? If so whats wrong with `Something::s_value{5};` ?

Edit: i think i get it, the static member variable inside the class are essentially forward declaration not definitions waiting to be initialized. So much like regular functions we need to still explicitly use the type e.g int when defining it outside the class.

0 Reply



Alex Author

Reply to [Albert](#)¹³ December 2, 2024 8:15 pm PST

Yup, what's inside the class is essentially a forward declaration. What's outside the class is the actual definition.

1 Reply



Ali

October 2, 2024 12:18 am PDT

"Do not put the static member definition in a header file (much like a global variable, if that header file gets included more than once, you'll end up with multiple definitions"

Does that happen even if we use `#ifndef` preprocessors ??? Why ?? That should avoid multiple declarations/definitions I think

0 Reply



Alex Author

Reply to [Ali](#)¹⁴ October 2, 2024 8:54 am PDT

Header guards (via `#ifndef`) only prevent a header from being included more than once into a single translation unit. It does not prevent a header from being included (once) into multiple different translation units.

2 Reply



RaveN

Reply to [Alex](#)¹⁵ October 13, 2024 3:16 am PDT

If we want to define a static member of a class template, its definition must be in the header file. Add this to the article as well, if I understood this correctly. Thank you for your lessons!

0 Reply



David

Reply to [RaveN](#)¹⁶ November 16, 2024 2:29 am PST

but then a contradiction arises in these sentences:

"Do not put the static member definition in a header file"

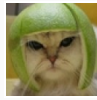
and then

"For template classes, the (templated) static member definition is typically placed directly underneath the template class."

And then what to do if the template class is in the header file?

👍 0

➡ Reply



Alex Author

➡ Reply to [David](#) ¹⁷ ⌚ November 17, 2024 9:40 pm PST

Thanks for pointing out the inconsistency. I clarified why the template case is acceptable (because the definition is implicitly inline).

👍 0

➡ Reply



Alex Author

➡ Reply to [RaveN](#) ¹⁶ ⌚ October 16, 2024 2:50 pm PDT

Added. Thanks!

👍 1

➡ Reply



dmigwi

⌚ August 22, 2024 11:36 am PDT

I think this statement is incorrect as used above. Is my interpretation wrong?

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

✎ Last edited 10 months ago by dmigwi

👍 1

➡ Reply



Alex Author

➡ Reply to [dmigwi](#) ¹⁸ ⌚ August 23, 2024 9:19 pm PDT

It's correct as written. The statement only applies to the definition of the static member, not further uses of it.

I added "(as definitions are not considered to be a form of access)." to the sentence to make it extra clear we're talking about the definition only.

👍 0

➡ Reply



Why Can't Const Static Class Members Be Initialize

⌚ July 31, 2024 5:47 pm PDT

Alex,

I have a question: Why Can't Const Static Class Members Be Initialized with Non-Constant Expressions in C++? and that's not the case for normal global or static variables?

👍 0

➡ Reply



Alex Author

Reply to [Why Can't Const Static Class Members Be Initialize](#) ¹⁹ ⌚ August 1, 2024 6:18 pm PDT

Stroustrup had this to say: "A class is typically declared in a header file and a header file is typically included into many translation units. However, to avoid complicated linker rules, C++ requires that every object has a unique definition. That rule would be broken if C++ allowed in-class definition of entities that needed to be stored in memory as objects."

This limitation was essentially lifted when inline variables were introduced into the language.

Normal global or static local variables don't typically have their definitions included into multiple translation units.

👍 1

➡ Reply

Links

1. <https://www.learncpp.com/author/Alex/>
2. <https://www.learncpp.com/cpp-tutorial/introduction-to-global-variables/>
3. <https://www.learncpp.com/cpp-tutorial/static-local-variables/>
4. <https://www.learncpp.com/cpp-tutorial/sharing-global-constants-across-multiple-files-using-inline-variables/>
5. <https://www.learncpp.com/cpp-tutorial/static-member-functions/>
6. <https://www.learncpp.com/>
7. <https://www.learncpp.com/cpp-tutorial/class-templates-with-member-functions/>
8. <https://www.learncpp.com/static-member-variables/>
9. <https://www.learncpp.com/cpp-tutorial/const-class-objects-and-const-member-functions/>
10. <https://gravatar.com/>
11. <https://www.learncpp.com/cpp-tutorial/static-member-variables/#comment-609632>
12. <https://www.learncpp.com/cpp-tutorial/static-member-variables/#comment-606440>
13. <https://www.learncpp.com/cpp-tutorial/static-member-variables/#comment-604730>
14. <https://www.learncpp.com/cpp-tutorial/static-member-variables/#comment-602619>
15. <https://www.learncpp.com/cpp-tutorial/static-member-variables/#comment-602632>
16. <https://www.learncpp.com/cpp-tutorial/static-member-variables/#comment-603028>
17. <https://www.learncpp.com/cpp-tutorial/static-member-variables/#comment-604209>
18. <https://www.learncpp.com/cpp-tutorial/static-member-variables/#comment-601155>
19. <https://www.learncpp.com/cpp-tutorial/static-member-variables/#comment-600359>
20. <https://g.ezoic.net/privacy/learncpp.com>