15.7 — Static member functions

👤 **ALEX**[1] 🕐 **OCTOBER 18, 2024**

In the previous lesson on 15.6 -- Static member variables (https://www.learncpp.com/cpp-tutorial/static-member-variables/)[2], you learned that static member variables are member variables that belong to the class rather than objects of the class. If a static member variable is public, it can be accessed directly using the class name and the scope resolution operator:

```cpp
#include <iostream>

class Something
{
public:
    static inline int s_value { 1 };
};

int main()
{
    std::cout << Something::s_value; // s_value is public, we can access it directly
}
```

But what if a static member variable is private? Consider the following example:

```cpp
#include <iostream>

class Something
{
private: // now private
    static inline int s_value { 1 };
};

int main()
{
    std::cout << Something::s_value; // error: s_value is private and can't be accessed directly outside the class
}
```

In this case, we can't access `Something::s_value` directly from `main()`, because it is private. Normally we access private members through public member functions. While we could create a normal public member function to access `s_value`, we'd then need to instantiate an object of the class type to use the function!

```
1   #include <iostream>
2
3   class Something
4   {
5   private:
6       static inline int s_value { 1 };
7
8   public:
9       int getValue() { return s_value; }
10  };
11
12  int main()
13  {
14      Something s{};
15      std::cout << s.getValue(); // works, but requires us to instantiate an object to
    call getValue()
16  }
```

We can do better.

Static member functions

Member variables aren't the only type of member that can be made static. Member functions can be made static as well. Here is the above example with a static member function accessor:

```
1   #include <iostream>
2
3   class Something
4   {
5   private:
6       static inline int s_value { 1 };
7
8   public:
9       static int getValue() { return s_value; } // static member function
10  };
11
12  int main()
13  {
14      std::cout << Something::getValue() << '\n';
15  }
```

Because static member functions are not associated with a particular object, they can be called directly by using the class name and the scope resolution operator (e.g. `Something::getValue()`). Like static member variables, they can also be called through objects of the class type, though this is not recommended.

Static member functions have no `this` pointer

Static member functions have two interesting quirks worth noting. First, because static member functions are not attached to an object, they have no `this` pointer! This makes sense when you think about it -- the `this` pointer always points to the object that the member function is working on. Static member functions do not work on an object, so the `this` pointer is not needed.

Second, static member functions can directly access other static members (variables or functions), but not non-static members. This is because non-static members must belong to a class object, and static member functions have no class object to work with!

Static members defined outside the class definition

Static member functions can also be defined outside of the class declaration. This works the same way as for normal member functions.

```cpp
#include <iostream>

class IDGenerator
{
private:
    static inline int s_nextID { 1 };

public:
    static int getNextID(); // Here's the declaration for a static function
};

// Here's the definition of the static function outside of the class.  Note we don't
use the static keyword here.
int IDGenerator::getNextID() { return s_nextID++; }

int main()
{
    for (int count{ 0 }; count < 5; ++count)
        std::cout << "The next ID is: " << IDGenerator::getNextID() << '\n';

    return 0;
}
```

This program prints:

```
The next ID is: 1
The next ID is: 2
The next ID is: 3
The next ID is: 4
The next ID is: 5
```

Note that because all the data and functions in this class are static, we don't need to instantiate an object of the class to make use of its functionality! This class utilizes a static member variable to hold the value of the next ID to be assigned, and provides a static member function to return that ID and increment it.

As noted in lesson 15.2 -- Classes and header files (https://www.learncpp.com/cpp-tutorial/classes-and-header-files/)[3], member functions defined inside the class definition are implicitly inline. Member functions defined outside the class definition are not implicitly inline, but can be made inline by using the `inline` keyword. Therefore a static member function that is defined in a header file should be made `inline` so as not to violate the One Definition Rule (ODR) if that header is then included into multiple translation units.

A word of warning about classes with all static members

Be careful when writing classes with all static members. Although such "pure static classes" (also called "monostates") can be useful, they also come with some potential downsides.

First, because all static members are instantiated only once, there is no way to have multiple copies of a pure static class (without cloning the class and renaming it). For example, if you needed two independent `IDGenerator`, this would not be possible with a pure static class.

Second, in the lesson on global variables, you learned that global variables are dangerous because any piece of code can change the value of the global variable and end up breaking another piece of seemingly unrelated code. The same holds true for pure static classes. Because all of the members belong to the class

(instead of object of the class), and class declarations usually have global scope, a pure static class is essentially the equivalent of declaring functions and global variables in a globally accessible namespace, with all the requisite downsides that global variables have.

Instead of writing a class with all static members, consider writing a normal class and instantiating a global instance of it (global variables have static duration). That way the global instance can be used when appropriate, but local instances can still be instantiated if and when that is useful.

Pure static classes vs namespaces

Pure static classes have a lot of overlap with namespaces. Both allow you to define variables with static duration and functions within their scope region. However, one significant difference is that classes have access controls while namespaces do not.

In general, a static class is preferable when you have static data members and/or need access controls. Otherwise, prefer a namespace.

C++ does not support static constructors

If you can initialize normal member variables via a constructor, then by extension it makes sense that you should be able to initialize static member variables via a static constructor. And while some modern languages do support static constructors for precisely this purpose, C++ is unfortunately not one of them.

If your static variable can be directly initialized, no constructor is needed: you can initialize the static member variable at the point of definition (even if it is private). We do this in the `IDGenerator` example above. Here's another example:

```cpp
#include <iostream>

struct Chars
{
    char first{};
    char second{};
    char third{};
    char fourth{};
    char fifth{};
};

struct MyClass
{
    static inline Chars s_mychars { 'a', 'e', 'i', 'o', 'u' }; // initialize static variable at point of definition
};

int main()
{
    std::cout << MyClass::s_mychars.third; // print i

    return 0;
}
```

If initializing your static member variable requires executing code (e.g. a loop), there are many different, somewhat obtuse ways of doing this. One way that works with all variables, static or not, is to use a function to create an object, fill it with data, and return it to the caller. This returned value can be copied into the object being initialized.

```cpp
#include <iostream>

struct Chars
{
    char first{};
    char second{};
    char third{};
    char fourth{};
    char fifth{};
};

class MyClass
{
private:
    static Chars generate()
    {
        Chars c{}; // create an object
        c.first = 'a'; // fill it with values however you like
        c.second = 'e';
        c.third = 'i';
        c.fourth = 'o';
        c.fifth = 'u';

        return c; // return the object
    }

public:
    static inline Chars s_mychars { generate() }; // copy the returned object into
s_mychars
};

int main()
{
    std::cout << MyClass::s_mychars.third; // print i

    return 0;
}
```

Related content

A lambda can also be used for this.

We show a practical example of this methodology in lesson 8.15 -- Global random numbers (Random.h) (https://www.learncpp.com/cpp-tutorial/global-random-numbers-random-h/#RandomH)[4] (though we do it with a namespace rather than a static class, it works the same way)

Quiz time

Question #1

Convert the `Random` namespace in the following example to a class with static members:

```cpp
#include <chrono>
#include <random>
#include <iostream>

namespace Random
{
    inline std::mt19937 generate()
    {
        std::random_device rd{};

        // Create seed_seq with high-res clock and 7 random numbers from
std::random_device
        std::seed_seq ss{
            static_cast<std::seed_seq::result_type>
(std::chrono::steady_clock::now().time_since_epoch().count()),
                rd(), rd(), rd(), rd(), rd(), rd(), rd() };

        return std::mt19937{ ss };
    }

    inline std::mt19937 mt{ generate() }; // generates a seeded std::mt19937 and
copies it into our global object

    // Generate a random int between [min, max] (inclusive)
    inline int get(int min, int max)
    {
        return std::uniform_int_distribution{min, max}(mt);
    }
}

int main()
{
    // Print a bunch of random numbers
    for (int count{ 1 }; count <= 10; ++count)
        std::cout << Random::get(1, 6) << '\t';

    std::cout << '\n';

    return 0;
}
```

Show Solution (javascript:void(0))[5]

6

7

2

8

Leave a comment...

Name*

Email*

Notify me about replies: 🔔

POST COMMENT

🐛 Find a mistake? Leave a comment above! ⑦

👤 Avatars from https://gravatar.com/[9] are connected to your provided email address.

219 COMMENTS

Newest ▾

Copernicus

🕐 June 5, 2025 12:22 am PDT

Question #1

```cpp
#include <chrono>
#include <random>
#include <iostream>

class Random
{
public:
    static int generate()
    {
        static std::random_device rd{};
        static std::seed_seq ss{ rd(), rd(), rd(), rd(), rd(), rd(), rd(), rd() };
        static std::mt19937 mt{ ss };
        static std::uniform_int_distribution dice{ 1, 6 };

        return dice(mt);
    }
};

int main()
{
    // Print a bunch of random numbers
    for (int count{ 1 }; count <= 10; ++count)
        std::cout << Random::generate() << '\n';

    std::cout << '\n';

    return 0;
}
```
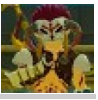
Not sure if this correct?

👍 0   ↪ Reply

sebby

March 31, 2025 4:48 pm PDT

static functions are more useful than you think, if you delete the default constructor, it is now a very useful namespace, with all the benefits of encapsulation. Here is a simple example:

```cpp
#include <iostream>

class Math {
private:
    static constexpr double pi{3.1415};
  public:
    static constexpr int max(int x, int y) {
        return (x > y) ?x: y;
    }
    static constexpr int add(int x, int y) {
        return (x + y);
    }
    static constexpr int min(int x, int y) {
        return (x < y) ? x: y;
    }
    static constexpr double circleArea(double r) {
        return pi * r * r;
    }
    Math() = delete;
};

int main() {
    std::cout << Math::max(3, 4) << '\n';
    std::cout << Math::min(5,6) << '\n';
    // Math obj{};  // explicitly deleted, object cannot be created
    std::cout << Math::circleArea(20) << '\n';


    return 0;
}
```

👍 0   ➤ Reply

> **PollyWantsACracker**
> 💬 Reply to sebby [10]   ⏱ May 15, 2025 9:35 am PDT
>
> whats the point of deleting constructor?
>
> 👍 0   ➤ Reply

sandersan

March 18, 2025 3:32 am PDT

I don't understand what practical significance this exercise has. Or is it just for testing purposes? Converting this namespace into a class does not seem to offer any improvements. Moreover, it feels semantically inconsistent. I believe it is better for the Random functionality to remain as a namespace as we had before.

May I ask what reasons there are, besides testing, to convert it into a class?

👍 0   ➤ Reply

Bassem
⏲ March 7, 2025 6:02 pm PST

```
1   #include <chrono>
2   #include <random>
3   #include <iostream>
4
5   class Random {
6   public:
7         static int get(int min, int max) { return std::uniform_int_distribution{ min,
    max }(generate()); }
8   private:
9         static std::mt19937& generate();
10  };
11  std::mt19937& Random::generate() {
12        std::random_device rd{};
13        std::seed_seq ss{
14              static_cast<std::seed_seq::result_type>
    (std::chrono::steady_clock::now().time_since_epoch().count()),
15                      rd(), rd(), rd(), rd(), rd(), rd(), rd() };
16        static std::mt19937 mt{ ss };
17        return mt;
18  }
19
20  int main() {
21      for (int count{ 1 }; count <= 10; ++count) { std::cout << Random::get(1, 6) <<
22  '\t'; }
      }
```

✎ *Last edited 3 months ago by Bassem*

👍 0   ↪ Reply

KLAP
⏲ January 16, 2025 3:01 pm PST

Quizz: Need to reread the whole chapter 8, even though I managed to convert the namespace into static classes, I don't quite get what the functions are doing.

```
1    #include <chrono>
2    #include <random>
3    #include <iostream>
4
5    class Random
6    {
7    public:
8        static std::mt19937 generate()
9        {
10           std::random_device rd{};
11
12           std::seed_seq ss{
13   static_cast<std::seed_seq::result_type>
     (std::chrono::steady_clock::now().time_since_epoch().count()),
14               rd(), rd(), rd(), rd(), rd(), rd(), rd()
15           };
16
17           return std::mt19937{ ss };
18       }
19
20       static int get(int min, int max)
21       {
22           return std::uniform_int_distribution{ min, max }(mt);
23       }
24
25   private:
26       static inline std::mt19937 mt{generate()};
27   };
28
29   int main()
30   {
31       // Print a bunch of random numbers
32       for (int count{ 1 }; count <= 10; ++count)
33           std::cout << Random::get(1, 6) << '\t';
34
35       std::cout << '\n';
36
37       return 0;
38   }
```

👍 0   ➥ Reply

Bhaskar Srivastava
🕐 December 22, 2024 3:48 am PST

Just a random pitfall I encountered while doing the quiz, apparently the files including "Random.h" (which I created during the past quizzes or practices having similar definitions in the questions) I was deleting from the codeblock IDE's file explorer was not actually deleting and were still linking to the current project, due to which my build was failing, I was debugging for past 30 mins, finally found the issue, Beware of that everyone...

👍 0   ➥ Reply

Konstantin
🕐 October 14, 2024 7:21 am PDT

Shouldn't the heading "Static member functions have no `*this` pointer" just be a `this`? As far as I understand, `*this` is not a pointer but the object itself. Can someone please explain?

👍 1   ➥ Reply

Alex `Author`
💬 Reply to Konstantin [12] 🕐 October 18, 2024 10:08 am PDT

Just a typo on my part. Section header corrected to remove the asterisk.

👍 0   ➥ Reply

Axactt
🕐 October 13, 2024 7:53 am PDT

As per example quiz: Does inline specifier inside the namespace variant also takes care of duration aspect which is associated with static keyword of class member variables? I mean creation of variable at program start and destruction at program end.
Or as the inline variables defined inside namespace are essentially global variables with external linkage/global scope, those inherently have static duration?

👍 0   ➥ Reply

Alex `Author`
💬 Reply to Axactt [13] 🕐 October 17, 2024 12:45 pm PDT

`inline` doesn't change the duration -- namespace scope variables (including those in the global namespace) have static duration.

👍 3   ➥ Reply

Baker Alshaif
🕐 May 4, 2024 6:38 am PDT

Compared to your solution, I did this. Why didn't you make all your functions inline?

```
1  class Random
2  {
3  public:
4      // Generate a random int between [min, max] (inclusive)
5      static inline int get(int min, int max);
6  private:
7      static inline std::mt19937 generate();
8
9      static inline std::mt19937 mt{ generate() }; // generates a seeded std::mt19937
   and copies it into our global object
10 };
```

👍 0   ➥ Reply

Alex `Author`

↩ Reply to Baker Alshaif [14] ⏱ May 8, 2024 10:46 am PDT

Functions defined inside a class are implicitly inline.

👍 14   ↪ Reply

Phargelm

⏱ April 30, 2024 6:11 am PDT

It looks like we can't call a static member function during initialization of a static member variable if that function wasn't defined yet. But in contrast it works with non-static members (we mentioned this in one of previous lessons).

```
class Test
{
private:
    static inline int x{ foo() }; // compile error: 'foo' was not declared in this
scope
    static int foo()
    {
        return 45;
    }
}
```

Non-static:

```
class Test
{
private:
    int x{ foo() }; // no issues
    int foo()
    {
        return 45;
    }
}
```

👍 2   ↪ Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/static-member-variables/
3. https://www.learncpp.com/cpp-tutorial/classes-and-header-files/
4. https://www.learncpp.com/cpp-tutorial/global-random-numbers-random-h/#RandomH
5. javascript:void(0)

6. https://www.learncpp.com/cpp-tutorial/friend-non-member-functions/
7. https://www.learncpp.com/
8. https://www.learncpp.com/static-member-functions/
9. https://gravatar.com/
10. https://www.learncpp.com/cpp-tutorial/static-member-functions/#comment-608940
11. https://www.learncpp.com/cpp-tutorial/static-member-functions/#comment-608639
12. https://www.learncpp.com/cpp-tutorial/static-member-functions/#comment-603109
13. https://www.learncpp.com/cpp-tutorial/static-member-functions/#comment-603070
14. https://www.learncpp.com/cpp-tutorial/static-member-functions/#comment-596646
15. https://g.ezoic.net/privacy/learncpp.com