# 15.7 — Static member functions



In the previous lesson on 15.6 -- Static member variables, 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:

```
#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:

```
#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!

```
#include <iostream>

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

public:
    int getValue() { return s_value; }
};

int main()
{
    Something s{};
    std::cout << s.getValue(); // works, but requires us to instantiate an object to
call getValue()
}
```

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:

```
#include <iostream>

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

public:
    static int getValue() { return s_value; } // static member function
};

int main()
{
    std::cout << Something::getValue() << '\n';
}
```

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.

```
#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, 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:

```
#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.

```
#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) (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:

```
#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