8.12 — Static member functions

BY ALEX ON SEPTEMBER 18TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 19TH, 2018

Static member functions

In the previous lesson on **static member variables**, you learned that static member variables are member variables that belong to the class rather than objects of the class. If the static member variables are public, we can access them directly using the class name and the scope resolution operator. But what if the static member variables are private? Consider the following example:

```
class Something
{
private:
    static int s_value;

};

int Something::s_value = 1; // initializer, this is okay even though s_value is private since it's a de
finition

int main()
{
    // how do we access Something::s_value since it is private?
}
```

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! We can do better. It turns out that we can also make functions static.

Like static member variables, static member functions are not attached to any particular object. Here is the above example with a static member function accessor:

```
class Something
{
private:
    static int s_value;
public:
    static int getValue() { return s_value; } // static member function
};

int Something::s_value = 1; // initializer

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

Because static member functions are not attached to a particular object, they can be called directly by using the class name and the scope resolution operator. 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!

Another example

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

Here's an example:

```cpp
class IDGenerator
{
private:
    static int s_nextID; // Here's the declaration for a static member

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

// Here's the definition of the static member outside the class.  Note we don't use the static keyword here.
// We'll start generating IDs at 1
int IDGenerator::s_nextID = 1;

// 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.

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 objects, this would not be possible with a single 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.

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:

```
class MyClass
{
public:
    static std::vector<char> s_mychars;
};

std::vector<char> MyClass::s_mychars = { 'a', 'e', 'i', 'o', 'u' }; // initialize static variable at poi
nt of definition
```

If initializing your static member variable requires executing code (e.g. a loop), there are many different, somewhat obtuse ways of doing this. The following code presents one of the better methods. However, it is a little tricky, and you'll probably never need it, so feel free to skip the remainder of this section if you desire.

```
class MyClass
{
private:
    static std::vector<char> s_mychars;

public:

    class _init // we're defining a nested class named _init
    {
    public:
        _init() // the _init constructor will initialize our static variable
        {
            s_mychars.push_back('a');
            s_mychars.push_back('e');
            s_mychars.push_back('i');
            s_mychars.push_back('o');
            s_mychars.push_back('u');
        }
    } ;

private:
    static _init s_initializer; // we'll use this static object to ensure the _init constructor is call
ed
};

std::vector<char> MyClass::s_mychars; // define our static member variable
MyClass::_init MyClass::s_initializer; // define our static initializer, which will call the _init cons
tructor, which will initialize s_mychars
```

When static member s_initializer is defined, the _init() default constructor will be called (because s_initializer is of type _init). We can use this constructor to initialize any static member variables. The nice thing about this solution is that all of the initialization code is kept hidden inside the original class with the static member.

Summary

Static member functions can be used to work with static member variables in the class. An object of the class is not required to call them.

Classes can be created with all static member variables and static functions. However, such classes are essentially the equivalent of declaring functions and global variables in a globally accessible namespace, and should generally be avoided unless you have a particularly good reason to use them.

8.13 -- Friend functions and classes

Index