8.12 — Static member functions

BY ALEX ON SEPTEMBER 18TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2016

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 definition

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 function 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 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 only access static member variables. They can not access non-static member variables. This is because non-static member variables 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:

```
class IDGenerator
{
private:
    static int s_nextID;

public:
     static int getNextID();
};

// We'll start generating IDs at 1
int IDGenerator::s_nextID = 1;

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

int main()
{
    for (int count=0; count < 5; ++count)
        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:

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

std::vector<char> MyClass::s_mychars = { 'a', 'e', 'i', 'o', 'u' }; // initialize static variable at point 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.

```cpp
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 called
};

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 constructor, 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 namespace, and should generally be avoided unless you have a particularly good reason to use them.

8.13 -- Friend functions and classes

Index

8.11 -- Static member variables

Share this:

Email Facebook 11 Twitter Google Pinterest

C++ TUTORIAL | PRINT THIS POST

38 comments to 8.12 — Static member functions

tony
December 14, 2007 at 7:12 am · Reply

Very good explanation and examples..

nageshrajmane
June 1, 2012 at 11:11 am · Reply

Grate explaination with accurate examples

San
December 27, 2009 at 8:06 am · Reply

The examples are simply simple to understand the concept…Thanks a lot

anon
February 2, 2010 at 4:01 am · Reply

Absolutely short and sweet !

peter
May 12, 2010 at 2:15 am · Reply

Hi Alex,

Did not understand the following….Can u explain?

"Second, 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 IDGenerators, this would not be possible."

What do u mean by two IDgenerators? Is it two classes?
Can some one explain the above paragraph in detail

4lgor1thm
May 12, 2010 at 5:29 am · Reply

Hi, Peter

"Second, 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)."

Static Classes can only have one instance derived from them (not an object, but a call to the static class itself.). Non-static classes can have an infinite number of instance derived from them (objects). Consider the example below; it's an excerpt from a small Console app I wrote in Visual C++ Studio Express 2010:

```
// StaticClasses.cpp : main project file.

#include "stdafx.h"

using namespace System;

//Our pure static class can only contain static items ...
static class StaticClass
{
private:
    static int s_nValue;

public:
    static int getValue(){return s_nValue++;}
};

class DynamicClass
{
private:
    int m_nValue;

public:
    DynamicClass(){m_nValue = 1;}
    int getValue(){return m_nValue++;}
};

int StaticClass::s_nValue = 0;

int main(array ^args)
{
    StaticClass cStaticClass;

    //cStaticClass::getValue(); //Throws an exception, cannot access static methods a
nd variables of an instantiated object of a static class

    //in the tutorial the author said it can be instantiated once
    //This is the reason why it holds on to its value even out of class scope.
    //Here we have one copy, the static class
    Console::WriteLine(L"StaticClass getValue call: " + StaticClass::getValue());
    Console::WriteLine(L"StaticClass getValue call: " + StaticClass::getValue());

    //in order to have multiple copies of the class, we need to create a non-static c
lass, and instantiate at least 2 objects
    DynamicClass cClass1;
    DynamicClass cClass2;

    //here 2 objects are being instantiated from a non-static class; here we have 2 c
opies, the same way we could use 2 different IDGenerators.
    Console::WriteLine(L"Instantiated Object 1: " + cClass1.getValue());
    Console::WriteLine(L"Instantiated Object 2: " + cClass2.getValue());

    Console::WriteLine(L"Press any key to continue ...");
    Console::ReadLine();
    return 0;
}
```

"… if you needed two independent IDGenerators, this would not be possible."

You can only derive one instance from a static class, and it is not an object.

I Hope this helps …

4lgor1thm

peter
May 12, 2010 at 11:37 pm · Reply

Thanks 4lgor1thm…:)

uma
May 26, 2010 at 10:26 pm · Reply

//cStaticClass::getValue(); //Throws an exception, cannot access static methods and variables of an instantiated object of a static class

this error i am not getting
cStaticClass.getvalue();---> i have written iike this i am not getting any error
cStaticClass is a object right so we can access member function or data of the claa by dot operator right.

y can u explain me in detail why i am not getting error

codingfreak
October 5, 2010 at 1:10 am · Reply

Seems STATIC classes are not supported under C++ … ??

When I tried to compile the above code I get error saying
"error: a storage class can only be specified for objects and functions"

Alex
January 23, 2016 at 2:53 pm · Reply

Yes, the static keyword can not be applied directly to a class. However, a class with all static member variables and static member functions is essentially a static class.

SUN
June 14, 2010 at 5:16 am · Reply

Hello Alex,

Can you please explain me what is the use of making a constructor static and when we need to do that ?

leon.li
June 30, 2010 at 9:38 pm · Reply

hi, SUN, I think static constructor is used for :
1.initiating some static member variables of a class;
2.if you don't write the static constructor, and your class has some static member variables which already given the initial values, the compiler will automated generate a static constructor for you.
3. I think in database singleton design mode, it is useful. Because we don't want to generate too much accessing thread for the database, it is waste the resource.

thanks.

SUN
July 6, 2010 at 8:58 am · Reply

Hi leon,

Thanks a lot for the information

Regards!

Alex
January 23, 2016 at 4:30 pm · Reply

C++ doesn't support static constructors. However, there are ways to work around that if you need to initialize static member variables in some complex way. I've updated the lesson to present one such method.

catraeus
April 2, 2011 at 5:11 pm · Reply

The whole of this example, with all of the static variables and methods, produces dangerous code. Threading becomes difficult (you have to build your own locks) and anybody can get to the variables from anywhere without the richness of the various c++ tools like boost, etc. This is completely done without problems, allowing the richness of the various other libraries that are aimed at OS management with a Singleton Pattern. I like the example (a href="http://www.yolinux.com/TUTORIALS/C++Singleton.html" title="here.">)

deajosha
February 26, 2012 at 10:55 pm · Reply

thanks a lot for sharing.

---

stheurkar
February 10, 2014 at 2:17 am · Reply

```
class IDGenerator
{
private:
static int s_nNextID;

public:
static int GetNextID() { return s_nNextID++; }
};

// We'll start generating IDs at 1
int IDGenerator::s_nNextID = 1;
```

How is it possible to access s_NextID defined under private access specifier in the line above.

> Alex
> January 23, 2016 at 4:32 pm · Reply
>
> int IDGenerator::s_nNextID = 1; is a definition, and as such, it's not subject to access controls.

Holosim
January 27, 2015 at 5:39 am · Reply

I'm confused. Just before you describe the use of public static member functions to access private static member variables, you mention that "we can't access Something::s_nValue directly from main(), because it is private." In the code snippets, however, on line 11 you initialize s_nValue to 1 directly. Why does this work? Shouldn't it fail since you are attempting to access a private member variable directly, even though it is static? Or does the limitation only apply inside a function, such as main()?

> Alex
> January 23, 2016 at 4:37 pm · Reply
>
> It works because s_value is a definition, and we're just providing an initializer. Because it's a definition, it's not subject to access controls.

Lorence
February 28, 2015 at 11:44 pm · Reply

<b>Hello alex,</b>

<p> Can you explain this to me? </p>

<p>Second, 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 IDGenerators, this would not be possible.</p>

Alex
January 23, 2016 at 4:43 pm · Reply

With normal classes, you can instantiate as many objects of the class as you need. Assuming we had a class named IDGen that used normal (non-static) member variables to generate IDs, we could do this:

```
IDGen one;
IDGen two;
std::cout << one.getNextID() << two.getNextID();
```

This would print “11”.

The ID generators are independent.

Now consider similar code using the IDGenerator class that has all static members that we defined in the lesson above:

```
IDGenerator one;
IDGenerator two;
std::cout << one.getNextID() << two.getNextID();
```

This would print “12”, because one and two are sharing the same static member variables that generate the IDs.

Piyoosh
April 2, 2015 at 9:05 pm · Reply

why static member function able to access private constructor

Please explane
class abc
{abc(){}
public:
static abc*instance;
staic abc* getinstance()
{
if(!instance)
    instance = new abc();
return instance

]
abc *abc::instance =NULL;
};

Alex
January 23, 2016 at 4:47 pm · Reply

Because the static function is a member of the class, just like a normal function.

puppi
June 23, 2015 at 10:56 am · Reply

hello everybody, i have a question.
in below code, why can i not be able to delete which previously i have been created instance? what’s wrong with my code?

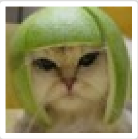
thanks a lot.

```
#include <iostream>
```

```

#ifndef NAMESPACE
    #define NAMESPACE
    using namespace std;
#endif

class Sample
{
private:
    int m_ID;
    char *szAdSoyad;

public:
    Sample()
    {

    }

    ~Sample()
    {
        // if there is any instance
        delete Sample::getInstance();
    }

    static Sample* getInstance()
    {
        static Sample *inst = 0;

        if (!inst)
        {
            inst = new Sample();
        }

        return inst;
    }
};

int main(void)
{
    Sample sn;
    delete &sn;

    system("pause");
    return 0;
}
```

Avneet
September 5, 2015 at 7:56 pm · Reply

Alex, Puppi's problem is mysterious to me too. Can you answer us..?
Thanks



Alex
September 7, 2015 at 1:12 pm · Reply

You can't explicitly delete variable sn because it hasn't been dynamically allocated.

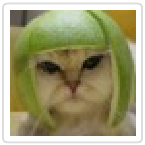
When sn goes out of scope, the destructor will naturally be called, which in this case will delete static variable inst, which is what you want anyway.

Avneet
September 7, 2015 at 5:34 pm · Reply

I removed that line + "pause"(may be it's a system call), put an output statement just after dynamically allocating memory like this:

```
inst = new Sample();
std::cout << "Allocation successful";
```

and a strange thing happens. This line get printed again and again like an infinite loop. I moved that line from there to destructor before calling
getInstance (), the same thing happens, it goes infinite. Then I placed that line of code below the call to getInstance () in destructor. Nothing printed and program crashes. Why????



Alex
September 7, 2015 at 6:48 pm · Reply

When sn goes out of scope, the destructor is called. The destructor calls Sample::getInstance(), which allocates an instance of Sample (which it really shouldn't be doing in a destructor, but lets put that aside) and then immediately deletes it (without setting inst back to null, so now inst is a dangling pointer). However, deleting it causes the destructor to be executed again. Since inst isn't null, it returns the dangling inst pointer, and tries to delete it again. This has undefined effects. In the best case, deleting it again will cause the destructor to be called again, re-deleting inst, causing the destructor to be called again, re-deleting inst, etc… An infinite loop.

In the worst case, it will crash.

In short, this program is extremely ill-formed. It's weird for a class to manage a static instance of itself. It leaves a dangling pointer after deleting the return value of getInstance(). And because inst is inside getInstance(), there's no way to set inst to null once it is deleted.

Avneet
September 7, 2015 at 7:17 pm

Thanks, confusion cleared

redhotchilialex
October 24, 2015 at 2:14 am · Reply

```
class IDGenerator
{
private:
    static int s_nNextID;

public:
     static int GetNextID() { return s_nNextID++; }
};

// We'll start generating IDs at 1
int IDGenerator::s_nNextID = 1;
```

s_nNextID is private but we can access it directly outside the class declaration. I didn't understand this…can anyone explain me, please?

Alex
October 25, 2015 at 12:55 pm · Reply

Static member variables are a little weird. There's only one instance shared across all instance of the class. For this reason, although the static member variable is declared inside the class, it is defined _outside_ of the class. For this reason, the initialization happens outside of the class too.

Note that because s_nNextID is private, you can't use it outside of the class -- this definition is purely for initialization purposes.

Gopal
November 9, 2015 at 3:49 am · Reply

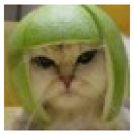
I think this is the only thing i have to memorize coz it is out of logic circle.

"There's only one instance shared across all instance of the class." -- This not seems logical reason.

As per my understanding if it is private, it must not change valve out side of class whether its static or not, but here it is happening. Its strange.

Even this below defination also seems strange coz we have to again apply type int to the definition even though we declared type at the time of definition inside of class.

int IDGenerator::s_nNextID = 1;



Alex
November 9, 2015 at 10:47 am · Reply

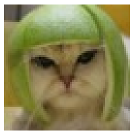
The access specifiers only restricts whether a variable can be accessed directly from outside the class or not. A static member is treated just like a normal member in this regard. The fact that there is only one instance of a static variable shared across all member of the class is irrelevant to how it is accessed.

The static member variable is _declared_ inside the class, but it needs to be initialized outside of the class, at the point where it is _defined_. I agree that the way it is initialized is a little funny looking.

sameer
November 19, 2015 at 10:33 pm · Reply

Hi,
what is use for "Static member functions",
what is different between "Static member functions" and "globa functions".

Thanks,
sameer



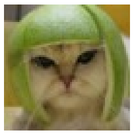
Alex
November 20, 2015 at 12:53 pm · Reply

Static member functions are primarily useful for accessing static member variables.

There isn't much difference between a static member function and a global function, outside of the fact that the static member function is associated with your class whereas a global function isn't. This means static member functions can access the private static data members of the class, whereas global functions can't.

Mekacher Anis
February 13, 2016 at 6:17 am · Reply

is std::cout a static class ?
I thought so because we only have one std::cout instance , a, I right ?



Alex
February 15, 2016 at 11:39 am · Reply

No, std::cout is a global variable of type std::ostream.