8.11 — Static member variables

BY ALEX ON SEPTEMBER 14TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 21ST, 2016

Review of static keyword uses

In the lesson on **file scope and the static keyword**, you learned that static variables keep their values and are not destroyed even after they go out of scope. For example:

```cpp
#include <iostream>

int generateID()
{
    static int s_id = 0;
    return ++s_id;
}

int main()
{
    std::cout << generateID() << '\n';
    std::cout << generateID() << '\n';
    std::cout << generateID() << '\n';

    return 0;
}
```

This program prints:

```
1
2
3
```

Note that s_id has kept its value across multiple function calls.

The static keyword has another meaning when applied to global variables -- it gives them internal linkage (which restricts them from being seen/used outside of the file they are defined in). Because global variables are typically avoided, the static keyword is not often used in this capacity.

Static member variables

C++ introduces two more uses for the static keyword when applied to classes: 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.

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

```cpp
class Something
{
public:
    int m_value = 1;
};

int main()
{
    Something first;
    Something second;

    second.m_value = 2;

    std::cout << first.m_value << '\n';
    std::cout << second.m_value << '\n';
```

```
16
17        return 0;
18    }
```

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 m_value: first.m_value, and second.m_value. first.m_nValue is distinct from second.m_nValue. Consequently, the program above prints:

```
1
2
```

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:

```cpp
1    class Something
2    {
3    public:
4        static int s_value;
5    };
6
7    int Something::s_value = 1;
8
9    int main()
10   {
11       Something first;
12       Something second;
13
14       second.s_value = 2;
15
16       std::cout << first.s_value << '\n';
17       std::cout << second.s_value << '\n';
18       return 0;
19   }
```

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), it turns out that static members exist even if no objects of the class have been instantiated! Much like global variables, they are created when the program starts, and destroyed when the program ends.

Consequently, it is better to think of static members as belonging to the class itself, not to the objects of the class. Because s_value exists independently of any class objects, it can be accessed directly using the class name and the scope operator (in this case, Something::s_value):

```cpp
1    class Something
2    {
3    public:
4        static int s_value; // declares the static member variable
5    };
6
7    int Something::s_value = 1; // defines the static member variable (we'll discuss this line bel
8    ow)
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;
      }
```

In the above snippet, s_value is referenced by class name 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.

Defining and initializing static member variables

When we declare a static member variable inside a class, we're simply telling the class that a static member variable exists (much like a forward declaration). Because static member variables are not part of the individual class objects (they get initialized when the program starts), you must explicitly define 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; // defines the static member variable
```

This line serves two purposes: it instantiates the static member variable (just like a global variable), and optionally initializes it. In this case, we're providing the initialization value 1. If no initializer is provided, C++ initializes the value to 0.

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.

If the class is defined in a .h file, the static member definition is usually placed in the associated code file for the class (e.g. Something.cpp). If the class is defined in a .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 compile error).

There is one exception where a static member definition line is not required: when the static member is of type const integer or const enum. Those can be initialized directly on the line in which they are declared:

```
1    class Whatever
2    {
3    public:
4        static const int s_value = 4; // a static const int can be declared and initialized directl
5    y
     };
```

In the above example, because the static member variable is a const int, no explicit definition line is needed.

An example of static member variables

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

```
1    class Something
2    {
3    private:
4        static int s_idGenerator;
5        int m_id;
6
7    public:
8        Something() { m_id = s_idGenerator++; } // grab the next value from the id generator
9
10       int getID() const { return m_id; }
11   };
12
13   // Note that we're defining and initializing s_idGenerator even though it is declared as priva
```

```
14    te above.
15    // This is okay since the definition isn't subject to access controls.
16    int Something::s_idGenerator = 1; // start our ID generator with value 1
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;
      }
```

This program prints:

```
1
2
3
```

Because s_idGenerator is shared by all Something objects, when a new Something object is created, the constructor grabs the current value out 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). This can really help when debugging multiple items in an array, as it provides a way to tell multiple objects of the same class type apart!

Static member variables can also be useful when the class needs to utilize an internal 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.
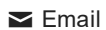
8.12 -- Static member functions

Index

8.10 -- Const class objects and member functions

Share this:

✉ Email   f Facebook 36   🐦 Twitter   G⁺ Google   P Pinterest