12.13 — Static member variables

ALEX AUGUST 29, 2021

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:

```
#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:

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

int main()
{
    Something first;
    Something second;

    first.m_value = 2;

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

    return 0;
}
```

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_value is distinct from second.m_value. Consequently, the program above prints:

```
2
1
```

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:

```
class Something
{
public:
    static int s_value;
};

int Something::s_value{ 1 };

int main()
{
    Something first;
    Something second;

    first.s_value = 2;

    std::cout << first.s_value << '\n';
    std::cout << second.s_value << '\n';
    return 0;
}
```

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 resolution operator (in this case, Something::s_value):

```
1  class Something
2  {
3  public:
4      static int s_value; // declares the static member variable
   };

   int Something::s_value{ 1 }; // defines the static member variable (we'll discuss this section
5  below)
6
7  int main()
   {
       // note: we're not instantiating any objects of type Something

       Something::s_value = 2;
       std::cout << Something::s_value << '\n';
       return 0;
8  }
```

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.

Best practice

Access static members by class name (using the scope resolution operator) rather than through an object of the class (using the member selection operator).

Defining and initializing static member variables

When we declare a static member variable inside a class, we're telling the compiler about the existence of a static member variable, but not actually defining it (much like a forward declaration). Because static member variables are not part of the individual class objects (they are treated similarly to global variables, and 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 linker error).

Inline initialization of static member variables

There are a few shortcuts to the above. First, when the static member is a const integral type (which includes char and bool) or a const enum, the static member can be initialized inside the class definition:

```
class Whatever
{
public:
    static const int s_value{ 4 }; // a static const int can be declared and initialized
directly
};
```

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

Second, static constexpr members can be initialized inside the class definition:

```
#include <array>

class Whatever
{
public:
    static constexpr double s_value{ 2.2 }; // ok
    static constexpr std::array<int, 3> s_array{ 1, 2, 3 }; // this even works for classes that support
constexpr initialization
};
```

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:

```
class Something
{
private:
    static int s_idGenerator;
    int m_id;

public:
    Something() { m_id = s_idGenerator++; } // grab the next value from the id generator

    int getID() const { return m_id; }
};

// Note that we're defining and initializing s_idGenerator even though it is declared as private above.
// This is okay since the definition isn't subject to access controls.
int Something::s_idGenerator { 1 }; // start our ID generator with value 1

int main()
{
    Something first;
    Something second;
    Something third;

    std::cout << first.getID() << '\n';
    std::cout << second.getID() << '\n';
    std::cout << third.getID() << '\n';
    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.

Next lesson

12.14 Static member functions

Back to table of contents

Previous lesson

12.12 Const class objects and member functions

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ coc

Name*

Email*

Avatars from https://gravatar.com/ are connected to your provided email address.

Notify me about replies: POST COMMENT

DP N N FOUT

Newest