15.6 — Static member variables

In the lesson 7.4 -- Introduction to global variables, we introduced global variables, and in lesson 7.10 -- Static local variables, we introduced static local variables. Both of these types of variables have static duration, meaning they are created at the start of the program, and destroyed at the end of the program. Such variables keep their values even if they go out of scope.

For example:

```
#include <iostream>

int generateID()
{
    static int s_id{ 0 }; // static local variable
    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 static local variable `s_id` has kept its value across multiple function calls.

Class types bring two more uses for the `static` keyword: 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.

Static member variables

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

```
#include <iostream>

struct Something
{
    int value{ 1 };
};

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

    first.value = 2;

    std::cout << first.value << '\n';
    std::cout << second.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 `value`: `first.value`, and `second.value`. `first.value` is distinct from `second.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:

```
#include <iostream>

struct Something
{
    static int s_value; // now static
};

int Something::s_value{ 1 }; // initialize s_value to 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), static members exist even if no objects of the class have been instantiated! This makes sense: they are created at the start of the program and destroyed at the end of the program, so their lifetime is not bound to a class object like a normal member.

Essentially, static members are global variables that live inside the scope region of the class. There is very little difference between a static member of a class and a normal variable inside a namespace.

Key insight

Static members are global variables that live inside the scope region of the class.

Because static member `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`):

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

int Something::s_value{ 1 }; // defines the static member variable (we'll discuss
this section below)

int main()
{
    // note: we're not instantiating any objects of type Something

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

In the above snippet, `s_value` is referenced by class name `Something` 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 using the class name and the scope resolution operator (::).

Defining and initializing static member variables

When we declare a static member variable inside a class type, 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 essentially global variables, you must explicitly define (and optionally initialize) the static member outside of the class, in the global scope.

In the example above, we do so via this line:

```
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 initializes it. In this case, we're providing the initialization value `1`. If no initializer is provided, static member variables are zero-initialized by default.

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 header (.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 source (.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).

Initialization of static member variables inside the class definition

There are a few shortcuts to the above. First, when the static member is a constant 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 defined and
initialized directly
};
```

In the above example, because the static member variable is a const int, no explicit definition line is needed. This shortcut is allowed because these specific const types are compile-time constants.

In lesson 7.9 -- Sharing global constants across multiple files (using inline variables), we introduced inline variables, which are variables that are allowed to have multiple definitions. C++17 allows static members to be inline variables:

```
class Whatever
{
public:
    static inline int s_value{ 4 }; // a static inline variable can be defined and
initialized directly
};
```

Such variables can be initialized inside the class definition regardless of whether they are constant or not. This is the preferred method of defining and initializing static members.

Because `constexpr` members are implicitly inline (as of C++17), static `constexpr` members can also be initialized inside the class definition without explicit use of the `inline` keyword:

```
#include <string_view>

class Whatever
{
public:
    static constexpr double s_value{ 2.2 }; // ok
    static constexpr std::string_view s_view{ "Hello" }; // this even works for
classes that support constexpr initialization
};
```

Best practice

Make your static members `inline` or `constexpr` so they can be initialized inside the class definition.

An example of static member variables

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

```
#include <iostream>

class Something
{
private:
    static inline int s_idGenerator { 1 };
    int m_id {};

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

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

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:

Because `s_idGenerator` is shared by all `Something` objects, when a new `Something` object is created, the constructor initializes `m_id` with the current value 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).

Giving each object a unique ID can help when debugging, as it can be used to differentiate objects that otherwise have identical data. This is particularly true when working with arrays of data.

Static members variables are also useful when the class needs to utilize a 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.

Only static members may use type deduction (`auto` and CTAD)

A static member may use `auto` to deduce its type from the initializer, or Class Template Argument Deduction (CTAD) to deduce template type arguments from the initializer.

Non-static members may not use `auto` or CTAD.

The reasons for this distinction being made are quite complicated, but boil down to there being certain cases that can occur with non-static members that lead to ambiguity or non-intuitive results. This does not occur for static members. Thus non-static members are restricted from using these features, whereas static members are not.