6.10 — Static local variables

ALEX JULY 4, 2021

The term `static` is one of the most confusing terms in the C++ language, in large part because `static` has different meanings in different contexts.

In prior lessons, we covered that global variables have `static duration`, which means they are created when the program starts and destroyed when the program ends.

We also discussed how the `static` keyword gives a global identifier `internal linkage`, which means the identifier can only be used in the file in which it is defined.

In this lesson, we'll explore the use of the `static` keyword when applied to a local variable.

Static local variables

In lesson 2.4 -- Introduction to local scope, you learned that local variables have `automatic duration` by default, which means they are created at the point of definition, and destroyed when the block is exited.

Using the `static` keyword on a local variable changes its duration from `automatic duration` to `static duration`. This means the variable is now created at the start of the program, and destroyed at the end of the program (just like a global variable). As a result, the static variable will retain its value even after it goes out of scope!

The easiest way to show the difference between `automatic duration` and `static duration` variables is by example.

Automatic duration (default):

```
#include <iostream>

void incrementAndPrint()
{
    int value{ 1 }; // automatic duration by default
    ++value;
    std::cout << value << '\n';
} // value is destroyed here

int main()
{
    incrementAndPrint();
    incrementAndPrint();
    incrementAndPrint();

    return 0;
}
```

Each time incrementAndPrint() is called, a variable named value is created and assigned the value of 1. incrementAndPrint() increments value to 2, and then prints the value of 2. When incrementAndPrint() is finished running, the variable goes out of scope and is destroyed. Consequently, this program outputs:

```
2
2
2
```

Now consider the static version of this program. The only difference between this and the above program is that we've changed the local variable from `automatic duration` to `static duration` by using the `static` keyword.

Static duration (using static keyword):

```
#include <iostream>

void incrementAndPrint()
{
    static int s_value{ 1 }; // static duration via static keyword.  This initializer is only executed once.
    ++s_value;
    std::cout << s_value << '\n';
} // s_value is not destroyed here, but becomes inaccessible because it goes out of scope

int main()
{
    incrementAndPrint();
    incrementAndPrint();
    incrementAndPrint();

    return 0;
}
```

In this program, because `s_value` has been declared as `static`, it is created at the program start.

Static local variables that are zero initialized or have a constexpr initializer can be initialized at program start. Static local variables with non-constexpr initializers are initialized the first time the variable definition is encountered (the definition is skipped on subsequent calls, so no reinitialization happens). Because `s_value` has constexpr initializer `1`, `s_value` will be initialized at program start.

When `s_value` goes out of scope at the end of the function, it is not destroyed. Each time the function incrementAndPrint() is called, the value of `s_value` remains at whatever we left it at previously. Consequently, this program outputs:

```
2
3
4
```

Just like we use "g_" to prefix global variables, it's common to use "s_" to prefix static (static duration) local variables.

One of the most common uses for static duration local variables is for unique ID generators. Imagine a program where you have many similar objects (e.g. a game where you're being attacked by many zombies, or a simulation where you're displaying many triangles). If you notice a defect, it can be near impossible to distinguish which object is having problems. However, if each object is given a unique identifier upon creation, then it can be easier to differentiate the objects for further debugging.

Generating a unique ID number is very easy to do with a static duration local variable:

```
int generateID()
{
    static int s_itemID{ 0 };
    return s_itemID++; // makes copy of s_itemID, increments the real s_itemID, then returns the value
in the copy
}
```

The first time this function is called, it returns 0. The second time, it returns 1. Each time it is called, it returns a number one higher than the previous time it was called. You can assign these numbers as unique IDs for your objects. Because `s_itemID` is a local variable, it can not be "tampered with" by other functions.

Static variables offer some of the benefit of global variables (they don't get destroyed until the end of the program) while limiting their visibility to block scope. This makes them safer for use even if you change their values regularly.

> **Best practice**
>
> **Initialize your static local variables. Static local variables are only initialized the first time the code is executed, not on subsequent calls.**

Static local constants

Static local variables can be made const. One good use for a const static local variable is when you have a function that needs to use a const value, but creating or initializing the object is expensive (e.g. you need to read the value from a database). If you used a normal local variable, the variable would be created and initialized every time the function was executed. With a const static local variable, you can create and initialize the expensive object once, and then reuse it whenever the function is called.

Don't use static local variables to alter flow

Consider the following code:

```
#include <iostream>

int getInteger()
{
 static bool s_isFirstCall{ true };

 if (s_isFirstCall)
 {
  std::cout << "Enter an integer: ";
  s_isFirstCall = false;
 }
 else
 {
  std::cout << "Enter another integer: ";
 }

 int i{};
 std::cin >> i;
 return i;
}

int main()
{
 int a{ getInteger() };
 int b{ getInteger() };

 std::cout << a << " + " << b << " = " << (a + b) <<
'\n';

 return 0;
}
```

Sample output

```
Enter an integer: 5
Enter another integer: 9
5 + 9 = 14
```

This code does what it's supposed to do, but because we used a static local variable, we made the code harder to understand. If

someone reads the code in `main()` without reading the implementation of `getInteger()`, they'd have no reason to assume that the two calls to `getInteger()` do something different. But the two calls do something different, which can be very confusing if the difference is more than a changed prompt.

Say you press the +1 button on your microwave and the microwave adds 1 minute to the remaining time. Your meal is warm and you're happy. Before you take your meal out of the microwave, you see a cat outside your window and watch it for a moment, because cats are cool. The moment turned out to be longer than you expected and when you take the first bite of your meal, it's cold again. No problem, just put it back into the microwave and press +1 to run it for a minute. But this time the microwave adds only 1 second and not 1 minute. That's when you go "I changed nothing and now it's broken" or "It worked last time". If you do the same thing again, you'd expect the same behavior as last time. The same goes for functions.

Suppose we want to add subtraction to the calculator such that the output looks like the following:

```
Addition
Enter an integer: 5
Enter another integer: 9
5 + 9 = 14
Subtraction
Enter an integer: 12
Enter another integer: 3
12 - 3 = 9
```

We might try to use `getInteger()` to read in the next two integers like we did for addition.

```
1  int main()
2  {
3    std::cout << "Addition\n";
4    int a{ getInteger() };
5    int b{ getInteger() };
6    std::cout << a << " + " << b << " = " << (a + b) <<
   '\n';
7
8    std::cout << "Subtraction\n";
     int c{ getInteger() };
     int d{ getInteger() };
9
10   std::cout << c << " - " << d << " = " << (c - d) <<
   '\n';
11
12   return 0;
13 }
```

But this won't work, the output is

```
Addition
Enter an integer: 5
Enter another integer: 9
5 + 9 = 14
Subtraction
Enter another integer: 12
Enter another integer: 3
12 - 3 = 9
```

("Enter another integer" instead of "Enter an integer")

`getInteger()` is not reusable, because it has an internal state (The static local variable `s_isFirstCall`) which cannot be reset from the outside. `s_isFirstCall` is not a variable that should be unique in the entire program. Although our program worked great when we first wrote it, the static local variable prevents us from reusing the function later on.

A better way of implementing `getInteger` is to pass `s_isFirstCall` as a parameter. This allows the caller to choose which prompt will be printed.

Static local variables should only be used if in your entire program and in the foreseeable future of your program, the variable is unique and it wouldn't make sense to reset the variable.

Best practice

Avoid `static` local variables unless the variable never needs to be reset.

Quiz time

Question #1

What effect does using keyword `static` have on a global variable? What effect does it have on a local variable?

Show Solution

Next lesson
6.11 Scope, duration, and linkage summary

Back to table of contents

Previous lesson
6.9 Sharing global constants across multiple files (using inline variables)

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