static

Introduction

Most of the time, you'll design classes so that any two instances of that class are independent. That is, if you have two objects `one` and `two`, changes to `one` shouldn't affect `two` in any way. However, in some circumstances you'll want to share data among all copies of a class. For example, perhaps you'll want all copies of a class to access a single global resource, or perhaps there's some initialization code you'd like to call when you create the very first instance of a class. C++ thus provides the `static` keyword to create class-specific information shared by all copies of that class. Like `const`, `static` has its nuances and at times can be a bit counterintuitive. This handout discusses `static` member functions, their relation to `const`, and `static` data members.

`static` Data Members

In C++, *static data members* are class-specific variables that are shared by each instance of that class. That is, if you have multiple instances of a class, each uses the same copy of that variable, so changes to static data members affect multiple objects.

The syntax for declaring static data members is a bit confusing since it exists in two steps – declaration and definition. For example, suppose you have the following class definition:

```
class MyClass
{
    public:
        /* Omitted */
    private:
        static int myStaticData;
};
```

Here, `myStaticData` is declared as a static data member with the `static` keyword. However, the line `static int myStaticData` *does not* actually create the variable `myStaticData`. Instead, it simply tells the compiler that there will be a variable called `myStaticData` declared at some point in the future. Thus, to initialize `myStaticData`, you will need to add another line to your program that looks like

```
int MyClass::myStaticData = 137;
```

There are several important points to note here. First, when declaring the variable, you must use the fully-qualified name `MyClass::myStaticData` instead of just `myStaticData`. Second, you do *not* repeat the `static` keyword during the variable declaration – otherwise, the compiler will think you're doing something completely different (see the "More to Explore" section). Finally, although `myStaticData` is declared `private`, you are still allowed to (and in fact, are required to) declare it outside of the class definition. The reason for this two-part declaration/definition for static data is a bit technical and has to do with where the compiler allocates storage for variables, so for now just remember that static data has to be separately declared and defined.

Static data members look just like regular data members in almost every aspect beyond declaration. Consider the following member function of `MyClass` called `doSomething`:

```
void MyClass::doSomething()
{
    myStaticData ++; // Modifies myStaticData for all classes
}
```

Nothing here seems all that out-of-the-ordinary and this code will work just fine. Note, however, that when you're modifying `myStaticData`, you are modifying a variable that any number of other instances of `MyClass` might be accessing. Thus it's important to make sure that you only use static data when you're sure that the information isn't specific to any one class.

To understand how static data members can be useful, let's consider an example case. Suppose that you're debugging some code and you're pretty sure that you've forgotten to `delete` all copies of a certain object you've allocated with `new`. Since C++ won't give you any warnings about this, you'll need to do the instance counting yourself, and you decide to use static data members to do the trick.

The number of active instances of a class is class-specific information that doesn't pertain to any specific instance of the object. This is the perfect spot to use static data members, so to do our instance counting, we'll insert the following declaration into our class definition:

```
private:
    static int numInstances;
```

We'll also define the variable outside the class as `int MyClass::numInstances = 0`.

We know that whenever we create a new instance of the class, the class's constructor will be called. This means that if we increment `numInstances` inside the class's constructor, we'll correctly track the number of instances of the class that have been created over time. Thus, we'll rewrite the `MyClass` constructor to look like

```
MyClass::MyClass()
{
    // All older initialization code
    numInstances++;
}
```

Similarly, we'll decrement `numInstances` in the destructor. We'll also have the destructor print out a message if this is the last remaining instance of the class so we can see how many instances are left:

```
MyClass::~MyClass()
{
    // All older cleanup code
    numInstances--;
    if(numInstances == 0)
        cout << "No more active instances!" << endl;
}
```

`static` Member Functions

Inside of member functions, a special variable called `this` acts as a pointer to the current object. Whenever you're accessing instance variables, you're really accessing the instance variables of the `this` pointer. But how does C++ know what value `this` refers to? The answer is subtle but important. Suppose you have a class `MyClass` with a member function `doSomething` that accepts two integer parameters. Whenever you invoke `doSomething` on a `MyClass` object using this syntax:

```
myInstance.doSomething(a, b);
```

It's as though you're actually calling

```
doSomething(&myInstance, a, b);
```

Where `doSomething` is prototyped as

```
void doSomething(MyClass *const this, int a, int b).
```

For almost all intents and purposes this is a subtle distinction that from your standpoint as a programmer should rarely be of interest. However, the fact that an N-argument member function is really an N+1 -argument free function can cause problems in a few places. For example, suppose you're writing a class `Point` that looks like this:

```
class Point
{
    public:
        bool compareTwoPoints(const Point &one, const Point &two)
        {
            if(one.x < two.x)
                return true;
            if(one.x > two.x)
                return false;
            return one.y < two.y;
        }
    private:
        int x, y;
};
```

If you have a `vector<Point>` that you'd like to pass to the STL `sort` algorithm, you'll run into trouble if you try to use this syntax:

```
sort(myVector.begin(), myVector.end(), Point::compareTwoPoints);
```

The problem is that the `sort` function expects a function that takes two parameters and returns a `bool`. However, you've provided a function that takes *three* parameters: two points to compare and an invisible "`this`" pointer. Thus the above code will generate an error.

In cases like this, you'll want to create a member function that doesn't have an invisible `this`. These functions are referred to as *static member functions* and are functions that act like free functions but that exist within the context of a class. Thus, if you changed the declaration of `compareTwoPoints` to read

```
static bool compareTwoPoints(const Point &one, const Point &two) { ... }
```

The call to `sort` would work since `compareTwoPoints` would no longer have a `this` pointer.

Because static member functions don't have a `this` pointer, you cannot access or modify any instance variables inside a static member function. However, you may refer to any static data members of the class, since they exist independently of any specific class instance.

Let's return to our earlier example about tracking the number of instances of an object currently in use. While it's nice that the destructor prints out a message when the last instance has been cleaned up, we'd prefer a more robust model where we can call a member function to check how many more copies of the class exist. Since this function should be independent of any class instance (that is, you don't need to have an object to check how many more remaining objects of that type exist), we'll make the function `getRemainingInstances` a static member function, as shown here:

```
class MyClass
{
    public:
        /* Constructor, destructor, etc. */
        static int getRemainingInstances();
    private:
        static int numInstances;
};

int MyClass::numInstances = 0;
```

Then the implementation of `getRemainingInstances` might look something like this:

```
void MyClass::getRemainingInstances()
{
    return numInstances;
}
```

As with static data, note that when defining static member functions, you omit the `static` keyword. Only put `static` inside the class declaration.

You can invoke static member functions either using the familiar `object.method` syntax, or you can use the fully qualified name of the function. For example, with the above example, we could check how many remaining instances there were of the `MyClass` class by calling `getRemainingInstances` as follows:

```
cout << MyClass::getRemainingInstances() << endl;
```

`const` and `static`

Unfortunately, the `const` and `static` keywords do not always interact intuitively. One of the biggest issues to be aware of is that `const` member functions can modify `static` data. That is, even if you're working with a `const` object, it is still possible that that object will change state between member function calls. For example, consider the following class:

```
class ConstStaticClass
{
    public:
        void constFn() const;
    private:
        static int staticData;
};

int ConstStaticClass::staticData = 0;
```

Then the following implementation of `constFn` is completely valid:

```
void ConstStaticClass::constFn() const
{
    staticData++;
}
```

Although you might think that this code is invalid since you're modifying the class's data inside a `const` member function, the above code will compile and run without any problems. The reason has to do with what `const` literally means. To see this, we'll have to revisit the `this` keyword.

Recall that `this` is a pointer to the current object in the context of a member function. Thus, whenever you writing code in a member function like `myInstanceVariable = 137`, the compiler treats it as though you wrote `this->myInstanceVariable = 137`. Inside a `const` member function, the compiler treats `this` as a pointer-to-`const`, so statements like `this->myInstanceVariable = 137` won't work since you're modifying the data of a pointer-to-`const`.

However, static data members are treated differently than regular instance variables because each instance of the class shares a single instance of the data. This means that access to a static data member doesn't go through the `this` pointer, so the fact that `this` is a pointer-to-`const` won't prevent you from modifying static data members. Again we return to the issue of bitwise versus semantic `const`ness. While it's legal to modify static data members in a `const` function, you should nonetheless make sure that `const` code isn't going to change the state of the current object.

Additionally, since `static` member functions don't have a `this` pointer, they cannot be declared `const`. In the case of `getNumInstances`, this means that although the function doesn't modify any class data, we still cannot mark it `const`. Thus when working with static member functions, make sure that if the function modifies class data, you somehow communicate that information to other programmers.

Integral Class Constants

There is one other topic concerning the interaction of `const` and `static` that can be a bit vexing – class constants. Suppose you want to make a constant variable accessible only in the context of a class. Thus, what you'd want is a variable that's `const`, so it's immutable, and `static`, so that all copies of the class share the data. It's legal to declare these variables like this:

```
class ClassConstantExample
{
    public:
        /* Omitted. */
    private:
        static const int MyConstant;
};

const int ClassConstantExample::MyConstant = 137;
```

However, since the double declaration/definition can be a bit tedious, C++ has a built-in shorthand you can use when declaring class constants of primitive types. That is, if you have a `static const int` or a `static const char`, you can condense the definition and declaration into a single statement by writing;

```
class ClassConstantExample
{
    public:
        /* Omitted. */
    private:
        static const int MyConstant = 137; // Condense into a single line
};
```

This shorthand is common in professional code. Be careful when using the shorthand, though, because some older compilers won't correctly interpret it.

More to Explore

While this handout discusses `static` in relation to classes, there are two other meanings of `static`. The first is *static linkage specification*, which means that the specified member function or global variable can only be accessed by functions defined in the current file. Another is *static local variables*, local variables whose values are preserved between function calls. Both are common in practice, so you should be sure to check a reference text for some more info on their subtleties.

Practice Problems

1. Write a class `UniquelyIdentified` such that each instance of the class has a unique ID number determined by taking the ID number of the previous instance and adding one. The first instance should have ID number 1. Thus the third instance of the class will have ID 3, the ninety-sixth instance 96, etc. Also write a `const`-correct member function `getUniqueID` that returns the class's unique ID.
2. The C header file `<cstdlib>` exports two functions for random number generation – `srand`, which seeds the randomizer, and `rand`, which generates a pseudorandom `int` between 0 and the constant `RAND_MAX`. To make the pseudorandom values of `rand` appear truly random, you can seed the randomizer using the value returned by the `time` function exported from `<ctime>`. The syntax is `srand((unsigned int)time(NULL))`. Write a class `RandomGenerator` that exports a function `next` that returns a random `double` in the range [0, 1). When created, the `RandomGenerator` class should seed the randomizer with `srand` only if a previous instance of `RandomGenerator` hasn't already seeded it.