The `static` Declarator for Classes

`static` data members

Consider the following class that models a sphere shape:

```cpp
class Sphere {
public:
  Sphere(double r) : radius{r}, volume{(4.0*PI*r*r*r)/3.0} {}
  double Radius() const { return radius; }
  double Volume() const { return volume; }
  void Radius(double r) { radius = r; volume = (4.0*PI*r*r*r)/3.0; }
private:
  double const PI {3.14159};
  double radius, volume;
};
```

One of the problems with the definition of class `Sphere` is that every object of type `Sphere` has a read-only data member `PI`. This means that every object of type `Sphere` is unnecessarily burdened with an extra 8 bytes. From a memory efficiency standpoint, this seems to be a high price to pay since data member `PI` is only required to compute the volume of a sphere. One option is to remove `PI` as a data member and instead define it as a global variable:

```cpp
namespace {
  double const PI {3.14159};
}

class Sphere {
public:
  // as before ...
private:
  double radius, volume;
};
```

The memory requirements of each object of type `Sphere` are reduced from 24 bytes to 16 bytes. However, a significant disadvantage of this approach is that the name `PI` is in the scope of the file in which class `Sphere` is defined.

A better approach is to specify `PI` as a `static` data member that is private to the implementation of class `Sphere`. Class data members that are declared `static` belong to the entire class, instead of to a specific instance of the class. More specifically, a `static` data member is defined only once and then shared by all instances [that is, objects] of the class. That means that if the `static` member gets changed, either by a user of the class or within a member function of the class itself, then all members of the class will see that change the next time they access the `static` member. Like any other data member, `static` members must be declared within their class and can be `public` or `private`.

In the implementation of class `Sphere`, we prefer that clients don't have access to `PI`. Further, the value of `PI` is immutable - it is always equivalent to π - and therefore, `PI` is declared as a private, read-only, static data member of class `Sphere`:

```
class Sphere {
public:
  // as before ...
private:
  double radius, volume;
  static double const PI;
};
```

Since `static` data members of a class exist outside any object, objects of type class `Sphere` will now contain two data members: `radius` and `volume`. Like global objects, `static` data members must be defined outside any function at file scope. That's because memory is allocated for `static` data members immediately when the program begins, at the same time global variables are initialized. Since `static` data members are not part of individual objects of class `Sphere`, they're not initialized by the class' constructors. Therefore, `static` data members are default-initialized when they're defined [for example, global objects of built-in types are initialized to zero] or can be explicitly initialized using initializers. The statement below on line 11 defines an object named `PI` that is a read-only `static` member of class `Sphere` and has type `double`. Note that when a `static` member is defined outside the class, the `static` keyword doesn't appear. The keyword appears only with the declaration inside the class body.

```
 1  // class is defined in sphere.h
 2  class Sphere {
 3  public:
 4    // as before ...
 5  private:
 6    double radius, volume;
 7    static double const PI;
 8  };
 9
10  // static data member must be defined and initialized in sphere.cpp
11  double const Sphere::PI {3.14159};
```

A big advantage of `static` data members is that they exist beyond a particular instance of a class, but do not extend into conflict with other `static` data members defined within other classes. The name `PI` defined in class `Sphere` doesn't conflict with other `PI`s defined in other classes.

`static` member functions

In addition to `static` member variables, C++ supports `static` member functions that are not bound to any object and do not have a `this` pointer. This means they can only access `static` data and call other `static` functions. One corollary to this is that a `static` member function can be invoked *without ever creating an object of the class*. For example, a public `static` member function can be defined to compute a sphere's volume:

```
class Sphere {
public:
  // same as before ...
  static double vol(double r);
private:
  double radius, volume;
  static double const PI;
};
```

As with any other member function, a `static` member function can be defined inside or outside of the class body. Here the `static` member function `vol` is defined outside the class body; notice that the function can use `static` members directly without the scope operator:

```
// in implementation file sphere.cpp
double Sphere::vol(double r) {
  return (4.0*PI*r*r*r)/3.0;
}
```

`static` member function `vol` can be accessed directly through the scope operator without ever instantiating an object of type `Sphere`:

```
std::cout << "volume: " << Sphere::vol(3.0) << "\n";
```

Static data members and member functions can be used to keep track of the current number of `Sphere`s that are currently instantiated. This is done by declaring a `static` data member `counter` in class `Sphere` and defining it with initial value `0`. Next, `counter` is incremented in every `Sphere` constructor and decremented in the destructor which must now be explicitly defined. A public `static` member function `ctr` is declared and defined to return the current value of `counter`:

```
// in sphere.h
class Sphere {
public:
  Sphere(double r)
    : radius(r), volume((4.0*PI*r*r*r)/3.0) { ++counter; }
  ~Sphere() { --counter; }
  double Radius() const { return radius; }
  double Volume() const { return volume; }
  void Radius(double r) { radius = r; volume = (4.0*PI*r*r*r)/3.0; }
  static double vol(double r);
  static int    ctr();
private:
  double radius, volume;
  static int counter;
  static double const PI;
};

// in sphere.cpp
int Sphere::counter {0};
double const Sphere::PI {3.14159};
double Sphere::vol(double r) { return (4.0*PI*r*r*r)/3.0; }
int Sphere::ctr() { return counter; }
```