Static (Class-wide) Members

David Kieras
Prepared for EECS 381, Fall 2004

Non-static (ordinary) member variables
Regular member variables of a class exist in every object. That is, when you declare a class and list the member variables, you are saying that every object in the class should have its own space in memory for each member variable.

```
class Thing {
public:
     int x;
     int y;
};
...
int main()
{
     Thing t1, t2
     ...
```

After the declaration of Things t1 and t2, each consists of a piece of memory holding two integers, so we have t1.x, t1.y, t2.x, t2.y. Each object has its own individual x and y variables.

Static member variables
Sometimes it is handy to have a member variable that is associated with the whole class instead of individual objects. Such a variable occupies its own piece of memory, by itself, instead of being part of an individual object. It is essentially a global variable, but its name is contained inside a class scope, so it goes with the class instead of being known everywhere in the program. Such a member variable can be made private to a class, meaning that only member functions can access it. A good name for this property would be something like "class-wide" or "whole-class", but unfortunately, the overworked keyword "static" was used instead, so we have "static member variables".

Here is an example of a Thing class where a static member variable, named "count", is used to keep a count how many Things exist. Every time a Thing is created, the constructor will increment the count; every time a Thing is destroyed, the destructor will decrement the count. Because the count variable is not part of individual objects, but exists for the entire class, it "lives" past the creation and destruction of individual objects, retaining its value while the individual objects come and go. Example:

```
class Thing {
public:
     Thing ()
          {count++;}  // there is now one more Thing
     ~Thing ()
          {count--;} // there is now one fewer Thing
     int x;
     int y;:
     static int count;
};
```

In this example, count is a public static member variable whose full name is Thing::count. Even if it was private, the constructor and destructor member functions can access it because they are member functions. Because count is static, it is a class-wide member variable, not a member

variable of the individual thing objects; there is **only one** Thing::count variable in the entire program.

You have to define and initialize a static member variable somewhere in your code, at the top level (outside of any function), the same way you define and initialize a global variable. The compiler will ensure that the initialization is done **before** your program starts executing at the main function. In this definition and initialization line, you provide the full name of the static member variable using the class scope operator: The best place to define the static member variable is near the beginning of the .cpp file that goes with the header file that contains the class declaration. For example in the file "Thing.h", we have the Thing class declaration, and then in "Thing.cpp" we would have:

```
#include "Thing.h"
other includes, etc.

int Thing::count = 0;     // define the static member variable

code for Thing class member functions
```

Actually, the "= 0;" is optional; by default static variables are initialized to whatever type of zero is appropriate, but including it is a customary way of making this definition more obvious.

At any point in your code, you can refer to a static member variable either in the context of an object, or just with the class scope operator, assuming that the code has access:

```
Thing t;
...
x = t.count;  // get the count
x = Thing::count; // get the count
```

You don't have to have an object involved because the static member variable belongs to the whole class. If you access the static member variable with an object, like in "t.count", the only role the object "t" plays is to inform the compiler what class the "count" variable belongs to - count is not part of the object, but belongs to the whole class of objects. Actually, accessing a static member variable through an object is not usually done; the class scope operator form is usually how it is done, or because the variable is normally private, through a static accessor function (see below).

Defining and initializing private static member variables

You can make a static member variable private, and often want to. But you can (and must) still define it the same way as shown above, even though it is private. For example, in Thing.h:

```
class Thing {
public:
     Thing ()
          {count++;}
     ~Thing ()
          {count--;}
     int x;
     int y;

private:
     static int count;
};
```

And then in Thing.cpp, we have the same thing as when Thing::count was public:

```
#include "Thing.h"
other includes, etc.

int Thing::count = 0;// define the private static member variable

code for Thing class member functions
```

Yes, this looks odd because the definition and initialization statement appears to be accessing a **private** member variable from **outside** the class. Yes, it is doing just that. But you have to be able to define the member variable *somehow*, even if it is private. So this exception to the access rules is allowed. But now you can no longer refer to the member variable in other ways because it is private:

```
Thing t;
...
x = t.count;  // error! count is private!
x = Thing::count; // error! count is private!
```

The usual solution is to use a static member function to access the private static variable.

Static member functions
A static member function is like a static member variable in that you can invoke it without an object being involved. Regular member functions can only be applied to an object and have the hidden "this" parameter.

A static member function is like an ordinary function, but its name is contained inside the class scope instead of being global to the whole program or translation unit. This allows you to keep a function associated with a class, and also to ensure that the function name doesn't conflict with some other function with the same name. And like a ordinary function, because it does not apply to an individual object, there is no "this" parameter. A key advantage is that because it is a member of the class, it has access to private member variables; naturally, this includes static member variables.

So, continuing the example, you can use a static member function as a reader for a private static member variable:

```
class Thing {
public:
     Thing ()
          {count++;}
     ~Thing ()
          {count--;}
     int x;
     int y;

     static int get_count() {return count;}

private:
     static int count;
};
```

If desired, you can also define the function outside of the class declaration, the same way as an ordinary member function.

In your code, call the function as follows:

```
Thing t;
...

x = t.get_count();     // get the count
x = Thing::get_count(); // get the count
```

The function get_count is a static member function that returns the value of the static member variable count. Because it is static, it can be called either with or without an object being involved. If it is called with an object, as in t.get_count(), the only role that object "t" plays is to inform the compiler what class get_count belongs to. As with static member variables, accessing a static member function through an object is not usually done. Normally, the class scope operator form is used to call a static member function.

An important property of a static member function is that you can't access the ordinary (non-static) members of the class from within the static member function - there is no individual object automatically involved (there is no "this" object). If you need your static member function to access the members of an object in its class, you have to pass in the object as a parameter of some sort, or provide some other access (e.g. through a global variable or static member variable). Because the static member function is a member of the class, it can then access all members of the passed-in object.

Summary
A static member variable:
• Belongs to the whole class, and there is only one of it, regardless of the number of objects.
• Must be defined and initialized outside of any function, like a global variable.
• It can be accessed by any member function of the class.
• Normally, it is accessed with the class scope operator. If it is private, use a static member function to read or write it.

A static member function:
• Is like an ordinary non-member function, but its scope is the class.
• It can access all members of an object in its class, but only if you make the object available, such as in a parameter - there is no "this" object.
• Normally, it is called with the class scope operator.

Tip: If you get a linker error complaining about an undefined static member variable, it means you forgot the line of code that defines and initializes it. Like ordinary member variables, the appearance of the variable in the class declaration is just a declaration; no actual variables in memory are created by the declaration. A static member variable thus comes into existence when you define it in the initialization line. This is a common error; after making it a few times, you'll recognize it!

Note: Standard C++ allows you to initialize a static member from within the class declaration (see Stroustrup, 10.4.6.2), but it only works for const int static member variables, and the initializing expression must be a constant itself. Because this special case is only occasionally useful, I've adopted a uniform policy of initializing static members outside of the class declaration, as described in this handout.