STATIC DATA MEMBERS IN CPP

When you precede a member variable's declaration with **static**, you are telling the compiler that only one copy of that variable will exist and that all objects of the class will share that variable. Unlike regular data members, individual copies of a **static** member variable are not made for each object. No matter how many objects of a class are created, only one copy of a **static** data member exists. Thus, all objects of that class use that same variable. All **static** variables are initialized to zero before the first object is created. When you declare a **static** data member within a class, you are *not* defining it. (That is, you are not allocating storage for it.) Instead, you must provide a global definition for it elsewhere, outside the class. This is done by redeclaring the **static** variable using the scope resolution operator to identify the class to which it belongs. This causes storage for the variable to be allocated. (Remember, a class declaration is simply a logical construct that does not have physical reality.)

To understand the usage and effect of a **static** data member, consider this program:

```
#include  <iostream>
using namespace std;
class shared {
static int a;
int b;
public:
void set(int i, int j) {a=i; b=j;}
void show();
} ;
int shared::a; // define a
void shared::show()
{
cout << "This is static a: " << a;
cout << "\nThis is non-static b: " << b;
cout << "\n";
}
int main()
{
```

```
shared x, y;
x.set(1, 1); // set a to 1
x.show();
y.set(2, 2); // change a to 2
y.show();
x.show(); /* Here, a has been changed for both x and y because a is shared by both objects. */
return 0;
}
```

This program displays the following output when run.

```
This is static a: 1
This is non-static b: 1
This is static a: 2
This is non-static b: 2
This is static a: 2
This is non-static b: 1
```

Notice that the integer **a** is declared both inside **shared** and outside of it. As mentioned earlier, this is necessary because the declaration of **a** inside **shared** does not allocate storage. *As a convenience, older versions of C++ did not require the second declaration of a* ***static*** *member variable. However, this convenience gave rise to serious inconsistencies and it was eliminated several years ago. However, you may still find older C++ code that does not redeclare* ***static*** *member variables. In these cases, you will need to add the required definitions.*

A **static** member variable exists *before* any object of its class is created. For example, in the following short program, **a** is both **public** and **static**. Thus it may be directly accessed in **main( )**. Further, since **a** exists before an object of **shared** is created, **a** can be given a value at any time. As this program illustrates, the value of **a** is unchanged by the creation of object **x**. For this reason, both output statements display the same value: 99.

```
#include <iostream>
using namespace std;
class shared {
public:
```

```
static int a;
} ;
int shared::a; // define a
int main()
{
// initialize a before creating any objects
shared::a = 99;
cout << "This is initial value of a: " << shared::a;
cout << "\n";
shared x;
cout << "This is x.a: " << x.a;
return 0;
}
```

Notice how **a** is referred to through the use of the class name and the scope resolution operator. In general, to refer to a **static** member independently of an object, you must qualify it by using the name of the class of which it is a member. One use of a **static** member variable is to provide access control to some shared resource used by all objects of a class. For example, you might create several objects, each of which needs to write to a specific disk file. Clearly, however, only one object can be allowed to write to the file at a time. In this case, you will want to declare a **static** variable that indicates when the file is in use and when it is free. Each object then interrogates this variable before writing to the file.

The following program shows how you might use a **static** variable of this type to control access to a scarce resource:

```
#include   <iostream>
using namespace std;
class cl {
static int resource;
public:
int get_resource();
void free_resource() {resource = 0;}
};
int cl::resource; // define resource
int cl::get_resource()
{
if(resource) return 0; // resource already in use
else {
resource = 1;
return 1; // resource allocated to this object
}
}
int main()
{
cl ob1, ob2;
if(ob1.get_resource()) cout << "ob1 has resource\n";
if(!ob2.get_resource()) cout << "ob2 denied resource\n";
ob1.free_resource(); // let someone else use it
if(ob2.get_resource())
cout << "ob2 can now use resource\n";
return 0;
}
```

Another interesting use of a **static** member variable is to keep track of the number of objects of a particular class type that are in existence.

For example,

```
#include <iostream>
using namespace std;
class Counter {
public:
static int count;
Counter() { count++; }
~Counter() { count--; }
};
```

```
int Counter::count;
void f();
int main(void)
{
Counter o1;
cout << "Objects in existence: ";
cout << Counter::count << "\n";
Counter o2;
cout << "Objects in existence: ";
cout << Counter::count << "\n";
f();
cout << "Objects in existence: ";
cout << Counter::count << "\n";
return 0;
}
void f()
{
Counter temp;
cout << "Objects in existence: ";
cout << Counter::count << "\n";
// temp is destroyed when f() returns
}
```

This program produces the following output.

```
Objects in existence: 1
Objects in existence: 2
Objects in existence: 3
Objects in existence: 2
```

As you can see, the **static** member variable **count** is incremented whenever an object is created and decremented when an object is destroyed. This way, it keeps track of how many objects of type **Counter** are currently in existence. By using **static** member variables, you should be able to virtually eliminate any need for global variables. The trouble with global variables relative to OOP is that they almost always violate the principle of encapsulation.

Source : http://elearningatria.files.wordpress.com/2013/10/cse-iii-object-oriented-programming-with-c-10cs36-notes.pdf