Lab 6

Friend Function & Classes, This Pointer, and static Variables

Friend Classes

C++ provides the friend keyword to do just this. Inside a class, you can indicate that other classes (or simply functions) will have direct access to protected and private members of the class. When granting access to a class, you must specify that the access is granted for a class using the class keyword:

```
friend class aClass;
```

Note that friend declarations can go in either the public, private, or protected section of a class--it doesn't matter where they appear. In particular, specifying a friend in the section marked protected doesn't prevent the friend from also accessing private fields.

Here is a more concrete example of declaring a friend:

```
class Node
{
   private:
   int data;
   int key;
   // ...
 friend class BinaryTree; // class BinaryTree can now access data directly
};
```

Now, Node does not need to provide any means of accessing the data stored in the tree. The BinaryTree class that will use the data is the only class that will ever need access to the data or key. (The BinaryTree class needs to use the key to order the tree, and it will be the gateway through which other classes can access data stored in any particular node.)

Now in the BinaryTree class, you can treat the key and data fields as though they were public:

```
class BinaryTree
{
   private:
   Node *root;
```

```
    int find(int key);
};
int BinaryTree::find(int key)
{
    // check root for NULL...
    if(root->key == key)
    {
        // no need to go through an accessor function
        return root->data;
    }
    // perform rest of find
```

Friend Functions

Similarly, a class can grant access to its internal variables on a more selective basis--for instance, restricting access to only a single function. To do so, the entire function signature must be replicated after the friend specifier, including the return type of the function--and, of course, you'll need to give the scope of the function if it's inside another class:

```
friend return_type class_name::function(args);
```

For instance, in our example Node class, we would use this syntax:

```
class Node
{
    private:
    int data;
    int key;
    // ...
    friend int BinaryTree::find(); // Only BinaryTree's find function has access
};
```

Now the find function of BinaryTree could access the internals of the Node class, but no other function in BinaryTree could. (Though we might want a few others to be able to!)

Note that when friends are specified within a class, this does not give the class itself access to the friend function. That function is not within the scope of the class; it's only an indication that the class will grant access to the function.

Functions which are friends of a class and are defined within the class definition are also by default inline.

Friend functions are not members of the class, and therefore the access attributes do not apply to them. They are just ordinary global functions with special privileges.

Let's suppose that we wanted to implement a friend function in the Box class to compute the surface area of a Box object.

Try It Out - Using a friend to Calculate the Surface Area

We can see how this works in the following example:

Example 8.1

```
// Creating a friend function of a class
#include <iostream>
using namespace std;

class Box            // Class definition at global scope
{
   public:
      // Constructor definition
      Box(double lv=1.0, double bv=1.0, double hv=1.0)
      {
         cout << endl << "Constructor called.";
         length = lv;                        // Set values of
         breadth = bv;                       // data members
         height = hv;
      }

      // Function to calculate the volume of a box
      double Volume()
      {
         return length * breadth * height;
      }

   private:
      double length;     // Length of a box in inches
      double breadth;    // Breadth of a box in inches
      double height;     // Height of a box in inches

   //Friend function
   friend double BoxSurface(Box aBox);
};

// friend function to calculate the surface area of a Box object
double BoxSurface(Box aBox)
{
   return 2.0*(aBox.length*aBox.breadth +
               aBox.length*aBox.height +
               aBox.height*aBox.breadth);
}

int main(void)
{
   Box Match(2.2, 1.1, 0.5); // Declare Match box
```

```
   Box Box2;                  // Declare Box2 - no initial values

   cout << endl
        << "Volume of Match = "
        << Match.Volume();

   cout << endl
        << "Surface area of Match = "
        << BoxSurface(Match);

   cout << endl
        << "Volume of Box2 = "
        << Box2.Volume();

   cout << endl
        << "Surface area of Box2 = "
        << BoxSurface(Box2);
   cout << endl;

   return 0;
}
```

How It Works

We declare the function BoxSurface() as a friend of the Box class by writing the function prototype with the keyword friend at the front. Since the BoxSurface() function itself is a global function, it makes no difference where we put the friend declaration within the definition of the class, but it's a good idea to be consistent when you position this sort of declaration. You can see that we have chosen to position ours after all the public and private members of the class. Remember that a friend function isn't a member of the class, so access attributes don't apply.

The definition of the function follows that of the class. Note that we specify access to the data members of the object within the definition of BoxSurface(), using the Box object passed to the function as a parameter. Because a friend function isn't a class member, the data members can't be referenced just by their names. They each have to be qualified by the object name in exactly the same way as they might in an ordinary function, except, of course, that an ordinary function can't access the private members of a class. A friend function is the same as an ordinary function except that it can access all the members of a class without restriction.

The example produces this output:

This is exactly what you would expect. The friend function is computing the surface area of the Box objects from the values of the private members.

Placing friend Function Definitions Inside the Class

We could have combined the definition of the function with its declaration as a friend of the Box class within the class definition - the code would run as before. However, this has a number of disadvantages relating to the readability of the code. Although the function would still have global scope, this wouldn't be obvious to readers of the code, since the function would be hidden in the body of the class definition, and particularly since the function would no longer show up in the ClassView.

The this pointer

Every object in C++ has access to its own address through an important pointer called **this** pointer. The **this** pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.

Friend functions do not have a **this** pointer, because friends are not members of a class. Only member functions have a **this** pointer.

Let us try the following example to understand the concept of this pointer:

```
#include  <iostream>
using namespace std;
class Box
{
  public:
    // Constructor definition
    Box(double l=2.0, double b=2.0, double h=2.0)
    {
      cout <<"Constructor called." << endl;
      length = l;
      breadth = b;
      height = h;
    }
    double Volume()
    {
      return length * breadth * height;
    }
    int compare(Box box)
    {
      return this->Volume() > box.Volume();
    }
  private:
    double length;     // Length of a box
    double breadth;    // Breadth of a box
    double height;     // Height of a box
};

int main(void)
{
  Box Box1(3.3, 1.2, 1.5);    // Declare box1
  Box Box2(8.5, 6.0, 2.0);    // Declare box2

  if(Box1.compare(Box2))
  {
```

```
    cout << "Box2 is smaller than Box1" <<endl;
  }
  else
  {
    cout << "Box2 is equal to or larger than Box1" <<endl;
  }
  return 0;
}
```

When the above code is compiled and executed, it produces following result:

```
Constructor called.
Constructor called.
Box2 is equal to or larger than Box1
```

Static keyword in C++

Static variables keep their values and are not destroyed even after they go out of scope. For example:

```
int GenerateID()
{
   static int s_nID = 0;
   return s_nID++;
}

int main()
{
   std::cout << GenerateID() << std::endl;
   std::cout << GenerateID() << std::endl;
   std::cout << GenerateID() << std::endl;
   return 0;
}
```

This program prints:

```
0
1
2
```

Note that s_nID has kept it's value across multiple function calls.

The static keyword has another meaning when applied to global variables — it changes them from global scope to file scope. Because global variables are typically avoided by competent programmers, and file scope variables are just global variables limited to a single file, the static keyword is typically not used in this capacity.

Static member variables

When we instantiate a class object, each object gets it's own copy of all normal member variables. Member variables of a class can be made static by using the static keyword. Static member variables only exist once in a program regardless of how many class objects are defined! One way to think about it is that all objects of a class share the static variables. Consider the following program:

```
class Something
{
public:
    static int s_nValue;
};

int Something::s_nValue = 1;

int main()
{
    Something cFirst;
    cFirst.s_nValue = 2;

    Something cSecond;
    std::cout << cSecond.s_nValue;

    return 0;
}
```

This program produces the following output:

```
2
```

Because s_nValue is a static member variable, s_nValue is shared between all objects of the class. Consequently, cFirst.s_nValue is the same as cSecond.s_nValue. The above program shows that the value we set using cFirst can be accessed using cSecond!

Although you can access static members through objects of the class type, this is somewhat misleading. cFirst.s_nValue implies that s_nValue belongs to cFirst, and this is really not the case. s_nValue does not belong to any object. In fact, s_nValue exists even if there are no objects of the class have been instantiated!

Consequently, it is better to think of static members as belonging to the class itself, not the objects of the class. Because s_nValue exists independently of any class objects, it can be accessed directly using the class name and the scope operator:

```
class Something
{
public:
    static int s_nValue;
};

int Something::s_nValue = 1;

int main()
{
    Something::s_nValue  = 2;
    std::cout << Something::s_nValue;
    return 0;
}
```

In the above snippet, s_nValue is referenced by class name rather than through an object. Note that we have not even instantiated an object of type Something, but we are still able to access and use Something::s_nValue. This is the preferred method for accessing static members.

Initializing static member variables

Because static member variables are not part of the individual objects, you must explicitly define the static member if you want to initialize it to a non-zero value. The following line in the above example initializes the static member to 1:

```
int Something::s_nValue = 1;
```

This initializer should be placed in the code file for the class (eg. Something.cpp). In the absense of an initializing line, C++ will initialize the value to 0.

An example of static member variables

Why use static variables inside classes? One great example is to assign a unique ID to every instance of the class. Here's an example of that:

```
class Something
{
private:
    static int s_nIDGenerator;
    int m_nID;
public:
    Something() { m_nID = s_nIDGenerator++; }
```

```
    int GetID() const { return m_nID; }
};

int Something::s_nIDGenerator = 1;

int main()
{
    Something cFirst;
    Something cSecond;
    Something cThird;

    using namespace std;
    cout << cFirst.GetID() << endl;
    cout << cSecond.GetID() << endl;
    cout << cThird.GetID() << endl;
    return 0;
}
```

This program prints:

```
1
2
3
```

Because s_nIDGenerator is shared by all Something objects, when a new Something object is created, it's constructor grabs the current value out of s_nIDGenerator and then increments the value for the next object. This guarantees that each Something object receives a unique id (incremented in the order of creation). This can really help when debugging multiple items in an array, as it provides a way to tell multiple objects of the same class type apart!

Static member variables can also be useful when the class needs to utilize an internal lookup table (eg. to look up the name of something, or to find a pre-calculated value). By making the lookup table static, only one copy exists for all objects, rather than a copy for each object instantiated. This can save substantial amounts of memory.

Note:
Before doing your lab exercises run the examples.
Exercises will be provided by the instructors in the lab.