

# G52OBJ Lab Exercises 3

Steven Bagley

## 1. Introduction

This week, we are going to consolidate our understanding of *Reference Counting* by using it to track the lifetime of objects. We shall be using the same implementation considered in lecture 13 which is based around using `CRCOBJECT` as a base-class for all objects that we want to reference count.

We are going to look at building a reference counted implementation of the **Observer** pattern. If you remember back to lecture eight, the **Observer** pattern lets one object called the *subject* tell any number of other objects, called *observers* about changes in its state.

This set of exercises is going to be less hands-on than previous one and so you'll be required to create more of the code yourselves. However, as a headstart, I'll remind you that you'll need to include the following at the top of your `.cpp` files:

```
#include <iostream>
#include <string>

using namespace std;
```

along with `#include` lines for the header files of any classes the file uses.

The exercises is divide into three parts, all based around an implementation of the **WeatherStation** observer example we saw in lecture eight, which can be found on the website (<http://www.eprg.org/G52OBJ/>) in the file `lab3.tar.gz`. The first part of the exercise briefly goes over the source code to ensure we are all on the same wavelength and understand how the code works. This code doesn't handle the objects lifetime at all even though all the objects have `CRCOBJECT` as a base-class. In the second part, you'll be expected to take the source code above and add calls to the `Retain()` and `Release()` methods in the required places to ensure that the objects are referenced counted correctly (and so that the objects are destroyed when necessary). Finally, in the third part we implement a new *observer* for the Weather Station.

## 2. Weather Station Overview

The Weather Station is split into three classes, and an interface<sup>†</sup>. Unfortunately, we cannot provide a full set of weather recording instruments to gather the data for you so `main.cpp` attempts to generate plausible looking data for the station, which it passes to the `CWeatherStation` object that represents the weather station. Two sample observer classes are provided: `CCurrentConditionsDisplay` which simply prints out the current data values and `CAverageTemperatureDisplay` which keeps and displays a running average of the temperature. Both of these observer classes inherit from `IWeatherObserver`.

### 2.1. IWeatherObserver interface

The `IWeatherObserver` interface is identical to what we saw in the lecture. It provides only one method `Update()` which is called on each *observer* by the `CWeatherStation` object (which in this example, is the *subject* part of the **Observer** pattern). Note again, the C++ syntax for declaring a method is abstract.

---

<sup>†</sup> A class where all its methods are abstract (or pure virtual in C++ parlance). We use these where we want to loosely couple two objects together. In this case, the Weather Station doesn't know anything about the implementation of its observers—other than they support the `IWeatherObserver` interface.

Also, we cheat slightly here since this ‘interface’ inherits the reference counting implementation from `CRCObject`. Although this means that the interface is no longer strictly a true interface, it means that we can easily reference count our observers. I would advise you to use this trick in your coursework where needed.

## 2.2. CWeatherStation class

The `CWeatherStation` class contains the implementation of the weather station and acts as the *subject* in the **Observer** pattern. Therefore, it maintains a collection of *observers* (in this case, objects that implement the interface `IWeatherObserver`). To simplify the implementation, we use a simple array to store these pointers. Ideally, you would want to use a more dynamic collection (such as a linked list) but that would require going into an unnecessary level of C++ detail for this course. This is defined in the classes private section like so:

```
private:
    static const int kMaxObservers = 256;
    IWeatherObserver *m_observers[kNumObservers];
    int m_cObservers;
```

Note the first line just defines a constant `kNumObservers` that we can use when we want to refer to the size of the array. Since the actual size of the array is only defined in one place, we only have one place to alter it if we want to change its size.

The array itself is declared as an array of pointers to objects that support the `IWeatherObserver` interface. We also have an integer variable `m_cObservers` which we use to keep track of how many *observer* objects are in the array. We do not need to initialize the array at all, since we never look past `m_cObservers` in the array, so we just set the count of *observers* to be zero.

### 2.2.1. Supporting observers

The support for *observers* can be found in the three methods: `RegisterObserver()`, `RemoveObserver()`, and `NotifyObservers()`. The first of these, `RegisterObserver()` has the simplest implementation. First, we check that the incoming pointer to the observer does not equal `NULL` (this avoids any null pointer exceptions and yep, you’ve guessed it, C++ uses `NULL` where Java would use `andnull...`) `m_cObservers`, the count of observers, is less than `kMaxObservers`). If so, the pointer to the *observer* is stored in the array and the counter incremented.

`RemoveObserver()` is the opposite of `RegisterObserver()` in that it removes a previous registered *observer*. It does this by first checking every array entry between zero and the value of `m_cObservers` and if the pointer stored in the array is the same as the one to be removed then we clear that entry in the array (by setting it to `NULL`). This compares whether the two pointers point to the same physical object in memory. It *does not* compare whether the value of the two objects is the same (for that we’d need to implement a comparison method in the class). This implementation of `RemoveObserver()` is actually pretty nasty and complicates the job of `NotifyObservers()` and in reality we would probably use a different implementation, but it suffices for this example since we do not need to consider the implementation of a linked list or other abstract data type.

This leads us to the last method used to implement the *subject* side of the **Observer** pattern, `NotifyObservers()`. This method is relatively simple it steps over every entry in the `m_observers` array between zero and `m_cObservers` (we do not need to go over the whole array as we know from the implementation of `RegisterObserver()` that there cannot be a valid pointer in an array index greater than the value of `m_cObservers`). For each entry we step over, we call the `Update()` method on the object pointed to by that entry—passing the object’s state (as stored in its member variables) to the *observer*. However, we need to be careful as our implementation of `RemoveObserver()` leaves `NULL` pointers in the array. Therefore, the code contains a check for this case.

### 2.2.2. Implementing the WeatherStation

The final method in `CWeatherStation` is `SetMeasurements()`. This method is used by our `main()` method to simulate the data coming from the sensors. It just sets the member variables and then calls `NotifyObservers()` to fire off updates to all the *observers*.

### 2.3. The Observers

The two supplied *observer* objects themselves are incredibly simple. Both `CCurrentConditionsDisplay` and `CAverageTemperatureDisplay` inherit from `IWeatherObserver` (and so as we've already noted gain reference counting support from `CRCObject`) and both provide implementations of the `Update()` method. `CCurrentConditionsDisplay`'s `Update()` is incredibly simple, and just prints out the received values.

`CAverageTemperatureDisplay` is slightly more complex. Its `Update()` method keeps a running total of all the temperature values received (in `m_sumTemperature`) along with a count of the number of updates received (`m_cValues`). The `Display()` method then uses these two values to calculate the average temperature (so far) and prints it out.

### 2.4. Driving it all

Finally, we look at `main()` (found in `main.cpp`). This does two things, firstly it creates a `CWeatherStation` object and registers two observers with it (one of each type) using `RegisterObserver()`.

The second half of `main()` simulate a collection of weather sensors by generating random numbers within a restricted range using the `random()` function. We generate 25 random weather events (waiting a second between each one using `sleep()`) and use `SetMeasurements()` to tell the weather station about it. This in turn calls each *observer* that has been registered with it. When the system is compiled and run, you get messages from both observers telling you both the current values and also a running average temperature. Note, since the implementation `CRCObject` we are using (see below) generates a lot of debug information you may want to use one of the following unix commands to run the system (assuming you have compiled it into a file called `wstation`):

```
./wstation 2>/dev/null
./wstation > somefile
```

The first command redirects the debug information into `/dev/null`, so we don't have to look at it. The second redirects it into a file, so that we can look at the logged information later.

## 3. Adding Reference Counting

The provided implementation of the Weather Station is, as we saw above fully, working but it does not call any of the reference counting methods to handle the object's lifetime. Your task (should you chose to accept it) is to go through the code and add calls to `Retain()` and `Release()` to ensure the objects' lifetimes are managed correctly.

To make things slightly easier for you, I've made a couple of modifications to `CRCObject`. The first is what can only be described as getting close to a piece of witchcraft and is in the destructor for `CRCObject`. This is there for one reason only it makes sure that if you try and call methods on objects that have been deleted it'll crash the program there and then. This means that you'll be alerted to any mistakes you might make with the reference counting calls rather than having the code appearing to work and then crashing in a completely unrelated place. I'm not expecting you to understand how this works (and I most certainly *will not* be examining you on it) although I'll be happy to explain it if anyone is interested...

Secondly, if you look at the `CRCObject` class (which is used as the base-class for all the objects we want to reference count) you'll see that the implementation contains a bit more code than the version given in lecture 13. I've added code to output debug messages every time an object is `Retain()`ed and `Release()`d. In addition, it also keeps track of reference counting calls across the whole program and will print (to the error output) a number that if your program is correctly reference counting its objects should equal zero. So you can use this value to check that your program is working correctly. You'll be able

to see when this is the case, since the ‘Global Reference Count’ output at the end of the program will be zero if it is correct.

Some pointers are given below to help you add the calls in the correct places:

- Call `Retain()` before you store a pointer in a new variable.
- Call `Release()` when a pointer to an object goes out of scope (e.g. the end of a method/function for local variables and in an object’s destructor for member variables)
- If you are changing the value stored in a pointer (and that includes assigning `NULL` to the pointer), then remember to use the `Retain() - Release() - Copy` cycle. Retain the new pointer, release the current pointer and then set the variable with the new pointer value.
- Don’t forget you’ll need to implement a destructor for `CWeatherStation` that releases all the pointers in the `m_observers` array.

#### 4. A New Observer

The current implementation defines two observers: one that shows the current temperature, humidity and pressure values and a second that calculates a running average temperature.

In this part of the exercise, you are going to implement a new observer that keeps track of the minimum and maximum temperatures observed. The logic for this is very simple. Two member variables are stored in the observer, both `doubles`, that store the min. and max. temperature. As each new temperature are received by the observer (by a call to its `Update()` method), it compares the value with these variables and if it is lower and higher respectively then the value is stored in those variables.

Create a new class `CMinMaxObserver` that inherits from the interface `IWeatherObserver` (in the same way that all the weather observers do—you may find looking at the implementation of the `CAverageObserver` instructive). `IWeatherObserver` inherits from `CRCObject` and so your class will gain reference counting support automatically. Add two private member variables to store the min. and max. temperatures.

Now we need to add the `Update()` method to the class. The implementation for this is relatively simple. The new temperature is passed into the method in the `temperature` variable, and so you can compare this with the stored min. and max. values and update them if necessary. Finally, we print out a line informing the user of the min. and max. values similar to the following using `cout`.

```
Min. temperature recorded = 15oC, Max. temperature recorded = 28oC
```

That’s the observer itself implemented, all that remains is to tell the *subject* about it. If you look in `wstation.cpp`, you should be able to find the code that registers the other observers with the subject. Add code here to instantiate your object, and then in the same way that the other observers are registered, register your new observer. (Don’t forget to call `Release()` on the observers in the appropriate place, i.e. when this block of code has finished with its pointers).

Compile and run the program, and if all is well you should find that the program now reports the min. and max. temperatures detected as well the average and current values.

#### 5. Closing remarks

If you have any questions regarding the coursework or would like me to look over your answer then either catch me in the lab, or drop me an email to `srb@cs.nott.ac.uk`. In addition, I’ll upload my solution towards the end of the week that you can compare your solution too as well.