Faculty of Science, Engineering and Technology

Object Oriented Programming

Topic 8: Resource Management in C++

Resources

The following resources can help you with this topic:

- Interfaces
- UML Sequence Diagrams

Topic Tasks

Before starting to work on the tasks, first read through this entire document to get a sense of the direction in which you are heading. Complete the following tasks and submit your work to Doubtfire for feedback before the deadline (check Doubtfire for the submission deadline).

Remember to submit your progress, even if you haven't finished everything.

After you have **discussed** your work with your tutor and **corrected any issues**,
it will be signed off as complete.

Pass Task 18 - Robust Planet Rover

When using the C# programming language, you never had to worry about cleaning up after yourself because C# implements a convenient algorithm known as garbage collection.

Environmental scientists and performance conscious developers understand that convenient access to resources comes with a penalty. In applications that involve many objects, a program will quickly consume physical memory resources and start using virtual memory, which may involve exchanging blocks of memory with a slow device like a hard disk. At this point, the garbage collection algorithm may be activated, causing the program to be suspended for an undetermined amount of time while the mess is cleaned up.

Therefore, when real-time performance is critical (e.g. air traffic control, medical imaging, etc.) sophisticated development teams will use an object-oriented programming language like C++.

Because we did not pay any attention to memory management in Topic 9, the Planet Rover that you implemented using C++ likely suffers two problems

- memory leaks; and
- object aliasing.

In this exercise, we will write some unit tests that demonstrate these problems and then write the code required to correct them.

Memory Leaks

To investigate and eliminate memory leaks, first add a new base class named **Object** to your Planet Rover project. This class simply counts the number of instantiated objects as follows.

1. Edit **Object.h** to include the following *class declaration*

```
class Object {

public:

    Object();
    Object(const Object&);
    virtual ~Object();

    // return the count of current object instances
    static int getCount ();

private:

    // count of all object instances in existence
    static int _count;

};
```

As in C#, the `static` keyword is used to declare a class-wide field or method.

2. Similarly, edit **Object.cpp** to include the following *class definition*

```
/*
 * static (class-wide) members are effectively global variables
 * and are initialised accordingly
 */

int Object::_count = 0;

// provides read-only access to the instance count
int Object::getCount ()
{
    return _count;
}

Object::Object()
{
    _count ++;
}

Object::Object(const Object&)
{
    _count ++;
}

Object::~Object()
{
    _count --;
}
```

Note that the two constructors increment the instance count, and the destructor decrements it.

Note: The second constructor is known as the copy constructor. If A is a pointer to a Rover, the copy constructor enables a copy of A to be created as follows

```
Rover* B = new Rover (*A);
```

The copy constructor is one of the canonical methods; if the developer does not define it, then the C++ compiler will automatically generate a copy constructor that performs a shallow copy. The shallow copy is at the heart of the object aliasing problem that will be investigated in the next sub-section.

For now, don't pay any attention to the **&** (ampersand) character; we will not be using this feature of the C++ language.

3. With the **Object** class implemented, modify the **Rover**, **Battery** and **Device** classes such that they inherit **Object**.

Note: All of the children of the **Device** class will automatically inherit **Object**!

4. Now edit **TestRover.h** and **TestRover.cpp** to add the following unit test

```
void TestRover::testMemoryLeaks()
{
    Rover* rover = new Rover();
    rover->attachDevice( new Radar() );

    delete rover;

    CPPUNIT_ASSERT( Object::getCount() == 0 );

}
```

5. Run this test and watch it fail. Why? Is it reasonable to assume that, when a Rover is deleted, any devices that are attached to it should also be deleted? What if we wanted to delete a Rover but keep the devices? These questions start to probe the problems that arise when we take responsibility for cleaning up after ourselves.

Note: In this simple example, failure to delete the **Radar** would have only a small impact on the program's available memory resources. However, if the object was something large, like an image or a video, then failure to delete it would quickly consume available memory and cause the program to become unresponsive.

6. As when managing the environment, responsible use of resources requires adoption of a policy. When programming with C++, you must choose and implement an object ownership policy. In this simple example, we can choose between

    1. The Rover exclusively owns any Device that is attached to it and it deletes them when it is deleted;

    2. The Rover does not assume any responsibility for any Device that is attached to it and it does not delete them when it is deleted; or

    3. The Rover shares ownership of any Device that is attached to it and only the ones that are not currently in use by any other object will be deleted with the Rover is deleted.

7. Sharing ownership requires a smart pointer that implements reference counting; such a smart pointer is available in C++11 and will not be covered in this introductory unit. We currently have no other object that is responsible for managing the devices, so we will give the Rover exclusive ownership rights and responsibilities: implement the Rover class destructor and make it delete all of the attached devices, then verify that the Rover class passes all unit tests.

Note: If you have not done so already, you will have to add statements to delete any objects created in your previously written unit tests before this unit test will pass.

Object Aliasing

It is often useful to create a copy of an object, which is why the copy constructor is one of the canonical methods in C++. However, when a copy of a simple class like Rover is created, we are faced with another design question: should all of the devices be copied too? If so, how?

For this exercise, let's assume that the desired behaviour is to create a full copy of the Rover with copies of all of the devices. To start investigating this issue, let's create a demonstrative unit test:

8. Edit **TestRover.h** and **TestRover.cpp** to add the following

```
void TestRover::testCopyConstructor()
{
    Rover* rover = new Rover();
    rover->attachDevice( new Radar() );

    Rover* copy = new Rover(*rover);

    CPPUNIT_ASSERT( rover->deviceCount() == copy->deviceCount() );
}
```

9. Run this test and watch it fail. Why?

   Currently, the Rover copy constructor does nothing, but how to we implement it?

   Before considering the implementation, let's test the version that the compiler would generate automatically; this can be done by commenting out the declaration in **Rover.h** and the definition in **Rover.cpp** (this empty method was generated automatically by the NetBeans IDE and it is not identical to the version generated by the compiler).

Note: If the unit tests fail to compile after commenting out the copy constructor, try making a small change to **TestRover.cpp**, then undo the change and try compiling again. NetBeans tries to intelligently recompile only the code that has been changed, including code that depends on code that has been changed, and it sometime misses the dependency between a source code file (like TestRover.cpp) and a header file (like Rover.h).

   The compiler-generated copy constructor should pass our new unit test - is this ok?

10. We are now in a good position to discuss and investigate **object aliasing**.

    When the compiler automatically generates a copy constructor, it performs what is known as a shallow copy. In a shallow copy, any pointers to objects that are managed by object A are copied verbatim to object B, resulting in two objects (A and B) with the same set of pointers. If objects A and B implement an exclusive ownership policy, then they will both think that they own the objects to which they point and they will delete those objects when they are deleted.

See the problem? If A deletes all of the things to which it points, then B will be left with a bunch of pointers to objects that no longer exists.

Each pointer in B is now known as a **dangling reference**. If B attempts to use any of these objects (e.g. by calling one of their methods) then there will be a memory violation error and the program might crash. This is the main reason that developers become intimidated by the responsibility of memory management.

11. To see effect of a dangling reference, edit **TestRover.h** and **TestRover.cpp** to add the following unit test.

```
void TestRover::testObjectAliasing()
{
    Rover* rover = new Rover();
    rover->attachDevice( new Radar() );

    Rover* copy = new Rover(*rover);

    delete rover;

    // if this test fails, then the following line causes a crash
    delete copy;

    // if the program is still running, then the test has passed
    CPPUNIT_ASSERT (true);
}
```

On Mac and Linux, this unit test causes a memory violation error called a **segmentation fault**.

12. To allow **Rover** objects to be copied while maintaining the exclusive ownership policy, it is necessary to implement a **deep copy**. A deep copy does not make a verbatim copy of pointers; instead, it constructs new instances of the objects to which the pointer points.

    In the case of the **Battery** class, creating a copy is as easy as calling the **Battery** copy constructor. However, the **Device** class is *abstract*; therefore, it is not possible to construct a new instance of a **Device** - only the children that implement the operate method can be constructed. But how can the **Rover** class know the type of each **Device** in order to know which copy constructor to call? Radar? Solar Panel? Drill?

    In C#, we could use the `typeof(DerivedClass)` function to return a **Type** object that could then be used by the **Activator** to create an instance of **DerivedClass**.

    In C++, there is no similar language feature; instead, developers will typically use the Prototype Design Pattern, which consists of adding a pure virtual **clone** method to the desired base class. This method must be implemented by each of the derived classes, which simply return a new instance of the derived class type using its copy constructor.

13. Add a pure virtual method named **clone** to the **Device** base class and implement this method in each of the derived classes.
14. If a derived class has any attributes (or fields), then add unit tests to ensure that its copy constructor is properly initialising those fields.
15. Implement the copy constructor of the **Rover** class using the **Device::clone** method to perform a deep copy of all of the devices that are attached.
16. Demonstrate that your **Rover** class implementation passes all of the unit tests.
17. You now possess the resource management skills required to launch rockets, deploy robotic rovers, and blast things with lasers - well done!

Pass Task 18 - Assessment Criteria

Make sure that your submission demonstrates the following:

- The unit tests correctly verify that there are
  - no memory leaks,
  - no dangling references,
  - no object aliases, and
  - most importantly, no program crashes!
- The code follows a self-consistent coding convention and is well documented.
- The code compiles and the screenshot shows the tests passing.

Credit Task 2 - Case Study: Iterations 4 and 5

Note: Do not let Credit Tasks delay you in keeping up with unit due dates. If you are behind, skip these tasks and move on to the next Pass Task. You can always come back and have a go at this later if you get time.

Over the next few week you will implement a larger object oriented program that demonstrates the use of all of the concepts covered so far. This will help you develop a deeper understanding, and create additional pieces of work to communicate this understanding.

1. For this week complete Iterations 4 and 5.

Once your tests are working correctly add additional details to your cover page. Remember to relate what you are doing to the unit's learning outcomes.

Tip: Now would be a good point to reflect on anything you did in these iterations that was interesting or challenging. Elaborate on this in the cover page so that you can take advantage of this in your portfolio.

Credit Task 2 - Assessment Criteria

Make sure that your task has the following in your submission:

- Iteration 4 and 5 are implemented
- The new classes have XML documentation for all public features.
- Code must follow the C# coding convention used in the unit (layout, and use of case).
- The code must compile and the screenshot show the tests passing.

Distinction Task 2 - Custom Program UML Sequence Diagram

If you are aiming for a Distinction or High Distinction grade, you should demonstrate weekly progress on your custom to your tutor. This will help to ensure that your custom program will meet the Distinction criteria, and your tutor should be able to give you advice on how to structure your program.

This week, choose one of the most important and/or complicated algorithms or sequences of events in your custom program and draw a UML Sequence Diagram that describes the roles of the different objects and the order in which they send each other messages.

As with the UML Class Diagram, start by drawing the sequence diagram by hand, then scan or take a picture and send to your tutor for feedback.

Tip: Submit early! We don't expect a perfect design and we are here to help!

Distinction Task 2 - Assessment Criteria

Include the following in your submission:

- Cover sheet with a more detailed description of your Distinction project; discuss some of the subtleties and implementation details of your design, similar to the discussion of how the Battleships Game class manages each Player turn.
- UML sequence diagram of an important/complicated task that your program must perform.