

# NET2006: Object-Oriented Programming

## Final Project: The Battlefield Simulator

### Introduction

In previous lab work you created *Army* and *Actor* classes to manage objects in a battlefield simulator. The *Actor* base class is an *abstract base class* (meaning that no objects of type *Actor* are ever created). Several classes are derived from the *Actor* base class.

In this final project, you will use a stable version of code from the last *Actor-Army* lab (either my to-be-published solution or your own implementation). You will modify this code to implement the following:

- use an **STL** container to manage the *Actor* objects that are part of each *Army* object.
- Add coordinate attribute to *Actor* objects.
- Implement *Move()* behaviour for *Actor* objects.
- Implement a *Simulator* class that drives the whole simulation process.

You'll have the rest of the term to complete this project, though I will want to see milestones completed each lab session (Wednesday and Thursday).

You can choose to work on this final project individually or in pairs. If you need more practice to prepare for the final exam, you might consider working independently.

### The Details

#### STL Container

In your last iteration of the *Actor-Army* lab, you used a dynamically allocated array of pointers to *Actor* objects. The array of pointers was created using *new* after reading the first field of your data file (and the first field identified the number of *Actor* objects contained in the file). You ended up creating a container that was precisely sized for the number of *Actor* objects to be managed.

Of course, there's a potential future problem. Consider the following problem and the old-style solution . . . but don't be overwhelmed by all the steps . . . since these are the steps that you won't have to do. I want to clarify how much effort that intelligent use of the **STL** can save. Using an **STL** container avoids all these issues altogether.

Under the old approach, imagine that you wanted to add more *Actor* objects to an *Army*, and that addition exceeded the capacity of the existing array of pointers. You would need to:

- allocate a new array of pointers to *Actor* objects;
- copy the addresses of the existing *Actor* objects from the original array to the new array;
- release the old array;
- assign the address of the new array of pointers to *Actor* objects to the managing pointer in the *Army* class.

What a pain!

Use an **STL** container and you avoid having to undertake all the preceding steps. If, for example, you use a *vector* container, you

can continue to use *push\_back()* to add again and again without having to reallocate the *vector*. The *vector* object will automatically reallocate itself on an as needed basis. And it will reallocate itself in a highly efficient way. In its reallocation process, it will automatically copy any data from the original memory to the newly allocated memory. It automatically releases any old memory and realigns pointers.

When program execution is about to terminate, you do not need to concern yourself with releasing the hidden array that sits inside the *vector*. The *vector* object is an *automatic* object, thus, it is automatically released when it goes out of scope; and the *vector* destructor is automatically called to release the hidden array.

However, you still have some clean up to do.

You will still employ the *new* operator to create objects of differing *Actor* types: *Wizard*, *Elf*, *Hobbit* etc. And the rule about *new* and *delete* hasn't changed. For every use of the operator *new*, you must use a corresponding *delete* somewhere.

After you implement the conversion to an **STL**-based container, be sure to check for memory leaks using the Microsoft routines I showed you earlier.

In the preceding exploration, I have used the *vector* class as the example container, but you could use *list* or *set* as well. Essentially, you need a container that can support a *forward iterator*, since you'll need these to sweep through your containers later during processing.

#### Coordinate Attribute in Actor Objects

Since an *Actor* object can move around a two-dimensional environment, you'll need to track its position with a *Coordinate* value. During the run of the simulator, you'll likely need to find the distance between 2 *Actor* objects. Here's a suitable *Coordinate* class with a predefined overloaded operator to test the equivalence of two *Coordinate* values.

```

// FinalProject-Coordinate.hpp
#ifndef _FinalProject_Coordinate_hpp_
#define _FinalProject_Coordinate_hpp_

#include <iostream>
#include <iomanip>
using namespace std;

#include <cmath>

class Coordinate {
public:
    int nX;
    int nY;
public:
    Coordinate() : nX(rand()%20), nY(rand()%20) {}
    Coordinate(int x, int y) : nX(x), nY(y) {}
    int GetX() { return nX; }
    int GetY() { return nY; }
    bool North() { --nY; }
    bool South() { ++nY; }
    bool East() { ++nX; }
    bool West() { --nX; }
    void DisplayStatus() { cout << '(' << setw(2) << nX << '!' << setw(2) << nY << ')'; }
    bool operator==(const Coordinate& rcRHS)
    { return nX==rcRHS.nX && nY==rcRHS.nY; }
    double CalculateDistanceTo(const Coordinate& rcRHS)
    {
        int nDeltaX = nX-rcRHS.nX;
        int nDeltaY = nY-rcRHS.nY;
        return sqrt(double(nDeltaX*nDeltaX) + double(nDeltaY*nDeltaY));
    }
}; // end class Coordinate
#endif // _FinalProject_Coordinate_hpp_

```

## Move() Behaviour for Actor Objects

You need to decide how *Actor* objects will move. Since each derived *Actor* type may have slightly different capabilities, the *Move()* function is implemented as a virtual function. For example, if a *Wizard* is in possession of his staff, he may be able to teleport directly to an opponent he intends to fight, or to a wounded comrade he intends to defend. A *Hobbit* might move away from the nearest opponent. A *Human* might move toward an opponent if the *Human* has sufficient strength and health points, but move away from the nearest opponent if seriously outmatched (in terms of strength and health points). An *Orc*, perhaps, never shies from a fight and might always move towards the nearest opponent, no matter how wounded.

The preceding characteristics are *not* design specifications. I do not expect you to implement them as expressed above. I am providing them as examples of the different kinds of behaviours you should contemplate when specifying the algorithms for your *Move()* behaviours.

## Simulator Class

The *Simulator* class will bind all the elements together. The *Simulator* class must have access to the two *Army* objects. It must implement a main loop that drives the simulation.

Each cycle through the main loop can pick the next *Actor* object from one *Army* and tell it to *Move()*. Imagine that **Gandalf** is the

*Actor* object to be moved. Of course, the *Move()* function will need to know which is **Gandalf's** nearest opponent.

To find the nearest opponent, you'll need to:

- Iterate through all the *Actor* objects in the opposing *Army*.
- Calculate the distance to each opposing *Actor* object (remember Mr. Pythagoras).
- If the newly calculated distance is the new minimum hang onto that new minimum distance and the pointer to the corresponding opponent *Actor*.
- Upon exiting the loop, you'll have the minimum distance and pointer to the opposing *Actor* object that is closest.

Now that you know where to move **Gandalf**, you can call the *virtual Move()* function, sending the *Coordinate* of the opposing *Actor* as an argument. **Gandalf** will then move according to his rule set (which you've defined in your algorithm).

If, as a result of the move, two opposing *Actor* objects occupy the same *Coordinate*, then a battle unfolds. The battle can be determined by attributes in the base class only (to keep things a bit simpler).

## Submission

You will show your interim work at the start of each of your next lab periods (both Tuesday and Thursday, and we can use Friday as a work period as well).

### November 18

- STL implementation completed, tested, free of memory leaks.

### November 20

- Several algorithms (a description of the behaviour, not actual code) for movement of each derived class *Actor*. Each of these algorithms can be quite small, but you need to plan the behaviours of your *Actor* objects before you code them.
- One algorithm to determine the outcome of a battle (determined by the characteristics stored in the base class only).

### November 25

- *Coordinate* implementation in code (tested).
- Display of map showing the location of each *Actor* object based on the *Coordinate* stored in each *Actor* object.
- Optionally (at this point), also show the ability to move *Actor* objects on the screen.

### December 2

- The fully completed project will include the source code, the algorithms and static class diagrams showing the inheritance hierarchy of your *Actor* classes, their relationship to the *Army* and *Simulator* classes.