

# Object-Oriented Programming in C++

## Lab 10: Splitting Code and More Virtual Functions

### Introduction

Start with your solution<sup>1</sup> to Lab 9 and implement three changes:

- split the single .cpp file into several .cpp and .hpp files.
- derive two new classes from the **Actor** class.
- implement virtual functions for the derived classes.

### The Details: Splitting Code

#### Step 1: Build Existing Code

If necessary, download the solution to Lab 9 from my website and make sure that you can build a project that compiles the code correctly. Run the code with enough basic testing to be confident that the build completed correctly.

#### Step 2: Analyze the Code Before Splitting

I would highly recommend that you print the original code and pencil your planned splitting directly on the printout. You can split it however you want, but I'd suggest the following.

- **ActorArmy.hpp**: This header file will contain the class specifications for all **Actor** and **Army** classes. Initially, you'll only have one **Actor** class and one **Army** class, but you will be deriving a number different **Actor** types . . . but leave the derivation as a later step. Only inline member functions<sup>2</sup> will be resident in this file. All out-of-line member functions must be placed in the associated .cpp file.

Place appropriate **#if ! defined** and **#endif** preprocessor commands in your header file.

- **ActorArmy.cpp**: Place all out-of-line functions for the **Actor** and **Army** classes here.

Add required **#include** commands at the top of the file. In addition to things like `<iostream>` you'll need to include **"ActorArmy.hpp"**. Notice that `iostream` is surrounded by angle brackets `< >` and **ActorArmy.hpp** is surrounded by double quotes `""`. The `< >` cause the compiler to look to the standard system directories for the target file; the `""` cause the compiler to look in the current project directory.

- **main.cpp**: Obviously, the function `main()` will go here. But you'll probably want to add other things that are in direct support of `main()`, for example, the code to display the menu.

Add the required **#include** commands.

- **HelperFunctions.cpp**: There are several functions that help with programming, but are not members of any class. The *function definitions* for these go here. The likely candidate functions are:

- **RandomNumber()**
- **Get()** // int version
- **Get()** // array of char version

- **Wait()**

- **HelperFunctions.hpp**: This file contains the *function declarations*. Remember to place appropriate **#if ! defined** and **#endif** preprocessor commands in your header file.

#### Step 3: Add Newly Split Files to Project List

After you finish splitting the files, you'll need to tell the project to add these newly split files to the proper section of your project listing (of course, the project listing is available in the *Solution Explorer* window).

Add the .cpp files to the *Source* section. Add the .hpp files to the *Header* section.

#### Step 4: Compile and Test

If the preceding steps have been completed correctly, the program should compile and run exactly as before.

### The Details: Deriving Classes

#### Step 1: Create a Revised Data File

Here's a sample of some records in my simple test file. As you can see, two records are hobbits, one is a wizard. To exercise your program, you'll need to add several more.

```
3
h13Frodo Baggins 11 80 33 70
w17Gandalf The White 100 100 100 t
h15Samwise Gamgees 23 78 23 80
```

Notice that each record is preceded by a single character tag that identifies the record type. This single character will be read from the file; the character will be sent to the function **Actor::CreateNewActor()** which will create the correct object (**Hobbit** or **Wizard**) and return its address (as a base class pointer).

#### Step 2: Change to Virtual Functions

This is a very simple step. Add the **virtual** keyword to the functions that are to be handled as virtual functions:

- **Read()**
- **Write()**
- **Display()**

You'll need to add a pure **virtual** function in the **Actor** class called **GetSymbol()**. This function will be used to supply the tag character that will be used to re-write the actor records to disk (since each record will have a tag to identify its record type).

#### Step 3: Create a Hobbit Class

A workable **Hobbit** class is shown below. Hobbits are different from other actors because of their ability to move around unseen (which I'm capturing with a data member named **nStealth**). You can add it to your program without alternation.

<sup>1</sup> I will supply a working solution to Lab 9 later in the week. Feel free to use my solution if you had difficulty completing Lab 9.

<sup>2</sup> An inline function is one where both the function *declaration* and the function *definition* are inside the class specification. An out-of-line function is one where only the function *declaration* is inside the class specification. The function body (that is, its *definition*) is placed outside the class (with the correct scope resolution operator).

```

class Hobbit : public Actor {
private:
    int nStealth;
public:
    Hobbit() : nStealth(RandomNumber(30, 90)) {}
    virtual char GetSymbol() { return 'h'; }
    virtual bool Read(ifstream& rifsInput)
    {
        if ( ! Actor::Read(rifsInput) )
            return false;
        rifsInput >> nStealth;
        return true;
    }
    virtual void Write(ofstream& rofsOutput)
    {
        Actor::Write(rofsOutput);
        rofsOutput << ' ' << nStealth;
    }
    virtual void Display()
    {
        Actor::Display();
        cout << setw(15) << "Stealth:" << nStealth;
    }
};

```

## Submission

Include the following in your submission:

- *Memory Map*: Create a sample memory map that shows memory after adding at least two **Hobbit** objects and two **Wizard** objects. The diagram should clearly show the difference between the data members.
- *Test Plan* and *Test Files*: These two things work hand-in-hand. The *test plan* describes how you will test. The *test files* implement much of the testing.
- I also want a printout of your source code.

### Step 4: Create a Wizard Class

You need to work out the details for the **Wizard** class. You'll find many similarities between the approach to creating a **Hobbit** class and the **Wizard** class. But the new data member will not be an **int**; it will be a **bool** that tracks whether the actor is holding his magical staff (walking stick).

### Step 5: Add the CreateNewActor() Function

```

Actor* Actor::CreateNewActor(char cActorType)
{
    switch(cActorType) {
        case 'h': return new Hobbit;
        case 'w': return new Wizard;
        // other Actor types to be added later
        default: return NULL;
    }
}

```

The function *declaration* for **Actor::CreateNewActor()** must go in the class specification as a **static** function. The out-of-line implementation of the function (as shown above) will not use the keyword **static**; that keyword is only associated with the *function declaration*.

### Step 6: Modify the Army::Read() Function

One small change here. In Lab 9, you always created an **Actor** object (because there were no derived classes). In this lab, after reading the record tag (either 'h' for Hobbit or 'w' for Wizard) you'll use the **Actor::CreateNewActor()** to create the new object<sup>3</sup>.

### Step 7: Build Program and Test

Time to see if it works.

<sup>3</sup> With the use of a pure virtual function **GetSymbol()**, you are restricted from creating any objects of type **Actor**. The implementation of a pure virtual function converts the base class **Actor** into an *abstract class*, that is, no objects of type **Actor** will ever be created – only derived class objects (such as **Hobbit** or **Wizard**).