Homework 9

Homework 9 will involve converting a program from using raw C pointers, i.e., the C pointers you learned in C++, to using unique_ptr and shared_ptr. In the homework directory you will find the code for a Duck simulator using the Strategy pattern from last semester. You don't need to understand the strategy pattern to work this assignment, however.

You should convert the supplied program into three forms:

1. A program that only uses unique_ptr
2. A program that only uses shared_ptr and does initialization using

   std::shared_ptr<Type1>(new Type2(*args*))

   where Type2 ISA Type1
3. A program that only uses shared_ptr and does initialization using

   std::make_shared<Type1>(Type2(…))

   where Type1 is not abstract and Type2 ISA Type1. More on why Type1 should not be abstract below under "Some useful information".

Some useful information.

First bit of useful information:

When you have a statement that looks like an assignment but is doing an initialization, e.g.,

    Type1 var = std::move(unique_ptr<Type1>(new Type2(*args*)));

or

    var = std::move(unique_ptr<Type1>(new Type2(*args*)));

or

    Type1( ) : var(std::move(unique_ptr<Type1>(new Type2(*args*)))) {. . .}

where var is an object field and the statement is an initialization being performed in a constructor, C++ is actually doing an initialization and not an assignment. In these cases you do not have to use std::move(…), and the statements above could be correctly written as:

    Type1 var = unique_ptr<Type2>(new Type2(*args*));

    var = unique_ptr<Type2>(new Type2(*args*));

    Type1( ) : var(unique_ptr<Type1>(new Type2(*args*))) {. . .}

However, if you use std::move it will not be an error.

Second bit of useful information:

C++ tries to make the semantics of shared_ptr match the semantics of raw C pointers as closely as possible. It also tries to make the semantics of make_shared mimic the semantics of new as much as possible. As a result, if you code:

```
std::shared<AbstractBase> sp = std::make_shared<AbstractBase>(DerivedNotAbstract( ));
```

this mimicking new you will cause you to get an error since std::make_shared<AbstractBase> will be treated as an attempt to create a new AbstractBase object. The correct way to do this is to do:

```
std::shared<AbstractBase> sp = std::make_shared<DerivedNotAbstract>(DerivedNotAbstract());
```

For those of you that remember when I talked about the desire to program to interfaces, the use of the concrete DerivedNotAbstract twice should be bothersome. This can be avoided by having a factory method, std::shared_ptr<AbstractBase> factory(…) {…} and use it to do the initialization, e.g.,

```
std::shared<AbstractBase> sp = factory("Derived");
```

but this is not necessary for this homework.

What to turn in:

Turn in a directory *userid* with three subdirectories: Unique, Shared, MakeShared. Unique contains your program using only unique_ptr, Shared contains your program using only shared_ptr, and not using std::make_shared<Type>, and MakeShared contains the program that uses only std::make_shared<Type>(…) to make the shared pointers. If executing

```
g++ *.cpp
```

will compile your program, you do not need to include a make file.

Points:

1 points for compiling and running correctly

3 points for the Unique program to only use unique pointers

3 points for the Shared program to only use shared pointers and not use make_shared

3 points for the MakeShared program to only use shared pointers and to always use make_shared to create a new shared pointer.