CIS 15 Fall 2007, Assignment IV

Instructions

- This is assignment for Unit IV.
- It is worth 10 points.
- **It is due on Wednesday November 14** and must be submitted by email (as below).
- **Follow these emailing instructions:**
  1. Create a mail message addressed to **parsons@sci.brooklyn.cuny.edu** with the subject line **cis15 hw4**.
  2. Attach ONE **.zip** file containing all the **.cpp** source code files and the **.h** header files created below.
  3. Failure to follow these instructions will result in points being taken away from your grade. The number of points will be in proportion to the extent to which you did not follow instructions... (which can make it a lot harder for me to grade your work — grrrr!)

Program description

For this assignment, you will explore an aspect of programming that we have not discussed in class, but is important for learning to write clear, modular and reusable programs. It is called writing *multiple file programs*. The idea behind writing "multiple file programs" is that instead of writing a long program in one big file, you split the program up into a number of smaller files. The contents of each file is divided logically, so that you have the definition of one class in each file. By convention, each file is named with the name of the class whose definition is inside it, followed by the `.cpp` extension. In order to make the file's contents usable by other files, you also need to define a *header* file (named `<class>.h`) that contains the definition of the class name and member names, while the `.cpp` file contains the definition of the content of the class members. For example, here is how it would work for the `point` class:

```
// filename: point.h

class point {
private:
  int x, y;
public:
  int  getX() const;
  int  getY() const;
  void set( int x, int y );
  void print() const;
};
```

```
// filename: point.cpp
#include <iostream>
using namespace std;

#include "point.h"

int point::getX() const { return x; }

int point::getY() const { return y; }

void point::set( int x, int y ) {
  this->x = x;
  this->y = y;
}

void point::print() const {
  cout << "(" << x << ", " << y << ")" << endl;
}
```

You can compile `point.cpp` even though it doesn't have a `main()` function by invoking g++ with the `-c` switch, which says: compile but don't link. *Compiling* is the process of turning source code into object code (i.e., files that end in `.o`). *Linking* is the process of turning object code into executable code.
So for example, to compile but not link `point.cpp`, do:

```
unix-prompt> g++ -c point.cpp -o point.o
```

Next, we create a second source code file that will be used to test the point class, e.g.:

```
// filename: testpoint.cpp

#include <iostream>
using namespace std;

#include "point.h"

int main() {
  point p;
  p.set( 1, 2 );
  p.print();
}
```

which is compiled using:

```
unix-prompt> g++ -c testpoint.cpp -o testpoint.o
```

and linked together with point.o using:

```
unix-prompt> g++ testpoint.o point.o -o testpoint
```

and finally we can run the test program using:

```
unix-prompt> ./testpoint
```

A. Write multi-file programs and test programs for rabbit4.cpp

For each of the classes defined in the rabbit4.cpp program discussed in class and available on the class web page, create an individual C++ source (.cpp) file and header (.h) file and a test program:

1. point class (just like above) → point.cpp, point.h, testpoint.cpp
2. living class → living.cpp, living.h (NO TEST PROGRAM!)
3. plant class → plant.cpp, plant.h, testplant.cpp
4. carrot class → carrot.cpp, carrot.h, testcarrot.cpp
5. animal class → animal.cpp, animal.h, testanimal.cpp
6. rabbit class → rabbit.cpp, rabbit.h, testrabbit.cpp
7. fox class → fox.cpp, fox.h, testfox.cpp
8. world class → world.cpp, world.h, testworld.cpp

Do them in the order listed above. NOTE that you CANNOT create a test program for living.cpp because it is an abstract class. Remember, you cannot instantiate an object of an abstract class—you can only extend the abstract class, create definitions for its virtual function(s) and then instantiate objects of that type.

Note that you will need to include the header files for superclasses in the subclass definitions. In other words, since rabbit is a subclass of animal, you will need to include "animal.h" and "rabbit.h" in rabbit.cpp, and when you create a test program for rabbit.cpp, you will need to link both rabbit.o and animal.o, as well as the test program testrabbit.o to create an executable testrabbit.

While this may become tedious once you get the hang of it, being able to write code like this has distinct advantages of (1) "unit testing" each class as you write it and (2) creating modular classes that can be re-used without having to re-write code.

B. Put the pieces together for rabbit4.cpp's main

Now take the `main()` function from `rabbit4.cpp` and place it in a file called `hw4.cpp`. Make sure to include the necessary header files and then link this file along with all the superclass and subclass files to create the final executable:

```
unix-prompt> g++ -c hw4.cpp -o hw4.o
unix-prompt> g++ hw4.o point.o living.o plant.o
             carrot.o animal.o rabbit.o fox.o world.o -o hw4
```

C. Modify the program

Finally, create another class, called `frog`, that extends the `animal` class and defines its own `move()` function:

1. It should move by jumping a random number of x and y positions in a single move. Remember to check the wrap-around so the frog doesn't jump outside of the boundaries of the "world".
2. Create `frog.cpp`, `frog.h` and `testfrog.cpp`.
3. Modify `world.cpp` so that it instantiates a `frog` object, sets its initial location randomly, and roams around the world (*hint:* modify `rabbitRoam()`) preventing the rabbit and the fox from occupying the same location as the frog (but not participating in the "be eaten" ecosystem).

D. Marking rubric

This assignment is worth 10 points. The breakdown is as follows:

A cpp, header and test files for each class:
`living` = *1 point*
`plant` = *1 point*
`carrot` = *1 point*
`animal` = *1 point*
`rabbot` = *1 point*
`fox` = *1 point*
`world` = *1 point*

B hw4 main program file = *1 point*

C frog extension:
`frog.cpp` = *0.5 points*
`frog.h` = *0.5 points*
`testfrog.cpp` = *0.5 points* and
`world.cpp` modifications = *0.5 points*