cis15-fall2007-sklar, assignment II

instructions

- This is assignment for unit II.
- It is worth 10 points.
- **It is due on Monday October 11** and must be submitted by email (as below).
- **Follow these emailing instructions:**
  1. Create a mail message addressed to **sklar@sci.brooklyn.cuny.edu** with the subject line **cis15 hw2**.
  2. Attach ONLY the **.cpp** source code file 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 develop a program that simulates a robot moving around in a 2-dimensional world, looking for coins. The robot's current location is represented by an (x, y) point in space. Imagine that the robot has a controller that receives commands one at a time. Each command will tell it to move forward or turn 90 degrees, either clockwise or anti-clockwise. The program will have a broad *control loop*, and for each iteration of the loop, the robot must decide what to do (i.e., which command to execute). The goal is for the robot to locate all the coins hidden in its world. The robot's world is 10×10 units in size, and the x and y coordinates are in the range $[0, 9]$ (i.e., from 0 to 9, inclusive).

You will create 3 classes, each with multiple data and function members, and a `main()`. Each class is described in detail below, with step-by-step instructions for developing the various components of each class and testing the components individually. In the end, you'll write the `main()` and put all the pieces together—and this final product is what you'll submit.

The intermediary test programs are for your benefit as a developer and should not be submitted. Incidentally, these are called *unit tests* and are created to let you test and debug the individual units of a complex program. By the end of the semester, you should have gained the skills and experience to devise and construct your own unit tests, without me giving you step-by-step instructions.

A. read and store command-line input

The program will be named "**look-for**". It will receive input from the command line indicating the locations of the coins to look for, in the following form:

```
unix$ look-for <coin1-x> <coin1-y> <coin2-x> <coin2-y> <coin3-x> <coin3-y>
```

In other words, the program will take 6 input values, which are pairs of (x, y) coordinates for three coins hidden in the robot's world.

1. Create a `point` class, as we did in lecture on Monday Sep 24. The class should have two `int` data members: x and y. The class should have four function members:
   - `void print() const;`
     This function prints out the values of the data members x and y.

- `void set( int x, int y );`
  This function sets the values of the data members x and y. Note that since the function arguments are also called x and y, you should use the `this->` pointer to disambiguate between the class's data members and the function arguments.
- `int getX()`
  This function returns the value of the x data member.
- `int getY()`
  This function returns the value of the y data member.

2. Create a `main()` for testing that reads two command-line arguments: x and y; validates, stores and prints them.
   - Check that the user has entered 2 command-line arguments (remember that `argc` contains the number of command-line arguments, including the name of the C++ program you are running). If not, print an error message and exit the program.
   - Convert the command-line arguments to integers. *Hint:* use the `atoi()` function. If you are not familiar with this function, type `man atoi` at the Unix prompt.
   - Check that the arguments are within the range $[0, 9]$. If not, print an error message and exit the program.
   - Instantiate a `point` object (i.e., declare a variable of type `point`) and store the values of the command-line arguments in that object.
   - Echo the arguments (i.e., display them back to the user).
3. Modify your `main()` that you created above to read in 6 arguments from the command-line instead of two. Validate them, store each in a `point` variable and print them all. Again, this `main()` is also for unit testing.

B. establish the robot's world

1. Create a class called `world` that has one data member and two function members:
   - the data member is an array of three `point` objects, each one storing the location of one of the 3 coins the robot is looking for
   - `void print() const;`
     This function prints out the locations of the three coins.
   - `void set( int i, int x, int y );`
     This function sets the location of the i-th coin in the data member array to (x, y).
2. Modify your `main()` that you created in part A to instantiate a `world` object and store the validated and converted command-line arguments in the `point` array within the `world` object. Then call the `world` object's `print()` function to echo the input. Again, this `main()` is for unit testing.

C. define the robot

1. Create a `robot` class that has the following data and function members:
   - a `point` object to store the robot's current location in the world
   - an enumerated data type, called `orientation_type`, that defines the four directions that the robot could be facing (north, south, east or west), i.e., its "orientation"
   - a variable to store the robot's current orientation
   - `void init();`
     This function initializes the robot's current location to $(0, 0)$ and its current orientation to $east$.

- `void print() const;`
  This function prints the robot's current location and orientation in a pretty format, such as:
  `I am at (0,0) and I am facing east.`
- `void setOrientation( orientation_type orientation );`
  This function sets the value of the robot's orientation data member.
- `bool forward();`
  This function simulates the robot moving forward one step in the direction that it is facing. It checks to make sure that the robot is not at the edge of its world. It returns `true` if the robot moves forward successfully and `false` if the robot is at the edge of its world and cannot move forward.
- `void turnCW();`
  This function changes the robot's orientation, simulating a turn in the clockwise direction.
- `void turnAntiCW();`
  This function changes the robot's orientation, simulating a turn in the anti-clockwise direction.
- `bool eastEnd();`
  This function returns `true` if the robot has reached the east edge of its world.
- `bool westEnd();`
  This function returns `true` if the robot has reached the west edge of its world.
- `bool northEnd();`
  This function returns `true` if the robot has reached the north edge of its world.
- `bool southEnd();`
  This function returns `true` if the robot has reached the south edge of its world.
- `bool zag();`
  This function is called when the robot has been moving east and has reached the east edge of its world, in which case it should turn clockwise, go forward one step south and turn clockwise again (where it will be heading west for its next move).
- `bool zig();`
  This function is called when the robot has been moving west and has reached the west edge of its world, in which case it should turn anti-clockwise, go forward one step south and turn anti-clockwise again (where it will be heading east for its next move).*

2. Now modify the `main()` from the previous step to instantiate a `world` object. Define and perform some unit tests on each of the function members in the `world` class. For example, first call `init()` and then call `print()` to make sure the robot's position and orientation are correctly initialized. Or, call `init(); forward(); print();` to verify that `forward()` works as you expect. Do the same with `turnCW()`, `turnAntiCW()`, `zig()` and `zag()`. Again, this `main()` is for your testing, not for submission.

3. Now modify the `main()` so that the robot traverses its world, starting at $(0, 0)$ and ending at $(9, 9)$, by visiting every cell in $([0, 9], [0, 9])$ space (i.e., all 100 cells). Unit test this by printing out every cell visited, to make sure that the robot gets to all of them.

D. find the coins

Finally, you need to put in a check while the robot is traversing the world to see if it finds the coins. You make the design decision about how to do that. Every time the robot finds a coin, it should print out a message like:

```
I am at (2,0) and I found the first coin! I'm gonna be rich! Yippee!
```

Keep a count of how many moves the robot makes, and when the program exits, print a message saying how many moves the robot made.
Note: of course, traversing every cell in turn is not an optimal search strategy by any means, but we will refine it later in the semester. The exercise here is to develop classes of your own with data and function members.

E. marking rubric

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

- `point` class with two data members (x and y) and four function members (print(), set(), getX() and getY())
  (1 point)
- `world` class with one data member ($coins$) and two function members (print() and set())
  (1 point)
- `robot` class with two data members, enumerated type definition, and 12 function members (init(), print(), setOrientation(), forward(), turnCW(), turnAntiCW(), eastEnd(), westEnd(), northEnd(), southEnd(), zag() and zig())
  (3 points)
- handling of command-line input, including error handling
  (1 point)
- complete traversal of the world by the robot (visiting every cell until all coins are found)
  (1 point)
- determining if coins are found, reporting when they are found and keeping track of how many have been found, including stopping when the 3rd coin is found
  (2 points)
- counting and reporting the number of moves made by the robot
  (1 point)