Computer Science 229
Homework Project 1
2000 points

C++ Programming Project (Robots!)

Introduction

In this project you will write a simulator for a simple robot system and environment. You will also write a parser and generator for configuration files for the simulation. Lastly, you will write a number of programs, each of which provides a different algorithm to control the movements of a robot within a simulation. You will be provided with a simple utility that connects the simulation and all robots involved in a given execution.

There are several parts to this project which are summarized below. The order may not be the order they are released or the order of the checkpoint due dates.

a) Create a program that reads a configuration file and displays information about it.
b) Create a program that creates a configuration file, based on parameters given to the program.
c) Create a program that performs a simulation.
d) Create a program that controls a single robot to perform different tasks.
e) Create a complete make system for your project.
f) Create an OO Design Document to describe your project.

Simulation Environment

The simulation environment consists of a grid of squares with length and width to be specified by the configuration. (The configuration is read from a configuration file defined later.) The maximum size of the length or width is 35, and the minimum size is 1. The simulation environment may be rectangular (i.e. the width need not match the height). Each square in the simulation may contain objects and may also have properties.

Objects: Objects are items that may exist in a square in the simulation. Examples of potential objects are Robot, Rock, Block, Table, etc. The robot object is special and is described separately below.

Properties: Properties are conditions that may exist in a square and modify how objects act in or around that square, or that act on a robot in some way. Some examples of potential properties are Fog, Water, Darkness, Blinding Light, etc.

Simulation Objects

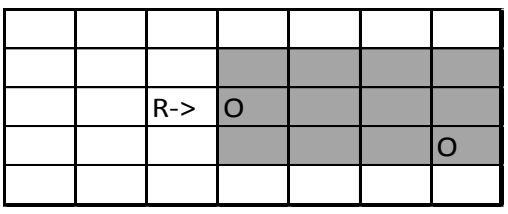
The following objects may be present in the simulation environment. These objects, their attributes, and how they affect the simulation are described below.

Robot Objects

These objects are sentient, and follow user-defined rules for movement and action.
The robot object has the following characteristics:

1. The robot object has a "direction" that defines the current orientation of the robot, and thus defines a front and back as well.
2. The robot object has a "battery" that contains its current energy, measured in positive integer units. The initial contents of the battery are defined in the configuration file. The battery recharges at a constant rate per turn, defined in the configuration file.
3. The robot may, on a turn, move forward one square at an energy cost defined in the configuration file.
4. The robot may, on a turn, turn 90 degrees clockwise or 90 degrees counterclockwise at an energy cost defined in the configuration file.
5. The robot may, on a turn, fire an energy pulse in the direction it is facing. An energy pulse has a single parameter, "strength", represented by e (energy). The cost of the energy pulse is configurable by the formula a + (b * e), where a and b are loaded from the configuration file. If an energy beam with energy e hits another robot, (c * e) energy is removed from the target robot, where c is loaded from the configuration file. Note that a, b, and c are real numbers (not necessarily integers) and that each robot has their own set of a, b and c parameters.
6. The robot may, on a turn, probe for "probe-able" objects. A probe has two parameters, "energy cost" and "beam width". The energy cost is deducted from the robot's battery when used. The beam width is an odd number that defines the width (measured in numbers of cells) in front of the robot to scan. For example, a beam width of 1 will scan the row or column directly in front of the robot. A beam width of 3 will scan the row/column immediately to the robot's left and right along with the center beam.
   Each beam returns the distance to the nearest probe-able object in its path, or 0 if there is no object in its path. The cost of using a single-beam probe is defined in the configuration file, and the cost of a multi-beam probe is the cost of a single-beam probe multiplied by the number of beams. See the figure for an example.
7. The robot may, on a turn, choose to do nothing.
8. The robot must perform exactly one action, chosen from #3-7, above.
9. The robot always "sees" the contents of the 8 squares surrounding it. All objects or properties, unless otherwise noted, are see-able by a robot.

Probe Example:



The cell marked 'R' contains the robot, facing towards the right. The cells marked 'O' contain probe-able objects. The shaded squares are the ones checked by a probe of beam width 3. The top beam would return 0, the middle would return 1, and the bottom would return 4.

Environment Objects

These objects are non-sentient, and follow simple movement rules:

- Objects do not move unless acted upon by an outside force.
- Unless otherwise noted, objects follow simple rules of physics. For example, if a movable object is pushed to the right, it will move one square to the right.
- Immovable objects, as indicated by their description, will not move, no matter how hard your robot tries.

The types of Objects are:

1. Earth Rock
   An Earth Rock represents a standard rock, placed in a square. Earth Rocks are immovable and probe-able.
2. Romulan Rock
   A Romulan Rock represents a rock with a built-in awesome cloaking device. Romulan Rocks are immovable but not probe-able, due to their built-in awesome cloaking device.
3. Ball
   A Ball represents a very large ball that robots may push around. Balls are movable and probable. When pushed by a Robot for n consecutive turns, the ball will continue to roll in that direction for an additional n squares, if possible. A Ball rolling on its own is stopped by any movable or immovable object, including another Ball.
4. Block
   A Block represents a very large block that robots may push around. Blocks are similar to balls except that if the robot stops pushing a Block, the Block immediately stops moving.
5. Energy Pill
   An Energy Pill represents an energy pill that adds charge to the first robot that eats it. Energy Pills are immovable and probe-able. Each Energy Pill has an associated amount of energy that it will restore if consumed.

Object Collisions

Two objects may not occupy the same square. A configuration file that has two objects in the same square may be considered in error.

If two Robots attempt to move into the same square, the result is dependent on which robot was moving. If one robot is moving and the other is stationary or turning, the stationary robot moves in the same manner that an Block would. If two robots attempt to move into the same square, neither robot moves and they both remain in their original squares.

When a Robot moves and the resulting move would result in a collision between two moveable objects, the second moveable object moves as well, per its movement rules defined above. This chain can be repeated with any number of moveable objects. Any immovable objects will stop the entire chain from moving. For more complicated collisions, such as if two Robots push two Blocks into the same square, choose intelligently and document clearly.

All Robots move at the same time, and thus do not allow any object to ‘phase’ through another by moving one Robot first.

Note that the edge of the simulation is not movable and objects cannot fall off.
The exception to this is Energy Pills. When a Robot enters a square with an Energy Pill, the Robot consumes the Energy Pill and gains the energy contained therein. The Energy Pill is destroyed in the process.

Simulation Properties

The following properties may be present in the simulation environment. These properties, their attributes, and how they affect the simulation are described below.

Environment Properties

The types of Properties are:

1. Lava
   A Lava property represents a pool of lava. Robots may traverse squares with lava pools at an extra energy cost to shield it from the lava. If the robot does not have the energy to protect it while crossing, the Robot is destroyed. Lava is not probe-able.
2. Water
   A Water property represents a pool of water. Robots may traverse squares with water pools at an extra energy cost to levitate over the water. If the robot does not have the energy to levitate while crossing, it is stuck in the water, and must wait until it gains enough energy to cross. Water is not probe-able.
3. Mud
   A Mud property represents a pool of mud. Robots may traverse squares with mud at an extra energy cost to levitate over the mud and an extra minimum number of turns to un-gunk the works. If the robot does not have the energy to levitate while crossing, it is stuck in the mud, and must wait until it does have the energy to cross. At that point it must still wait the extra number of turns to un-gunk. Mud is not probe-able.
4. Hole
   A Hole property represents a bottomless, gaping chasm. A Hole should be avoided at all costs, as anything that moves into or is pushed into a Hole is permanently removed from the simulation. A Hole is not probe-able.
5. Fog
   A Fog property represents fog that has descended onto a square. Fog has the unique ability to prevent a Robot from seeing (not probing) whatever is under the fog. Thus, when a Robot is immediately adjacent to a square containing Fog, the Robot cannot see what is under the Fog. The only thing reported to the Robot is the existence of Fog in that square. Fog does NOT prevent a Probe from determining if there is something under the Fog; however, un-probe-able objects under Fog are effectively invisible, and thus placing a Hole under Fog is particularly evil. Fog itself is not probe-able.
6. Jamming
   A Jamming property represents the existence of a radio jammer present in the square. A square that has the Jamming property will deflect a probe beam back to the source in such a way that it appears that the probe returns 0. Jamming is not probe-able.

Initial Configuration File

The initial configuration file specifies initial locations and attributes of objects and properties. The format of this file consists of the specification for each object and property using XML-like syntax. See the attached file for a sample file. Note that there is no requirement that the file be sorted in any manner, other than that the Simulation tag be first. Also note that not every possible tag is listed in the file for each possible object or property; however, every tag is listed at least once in the file. Any tag that can describe an object or property per its description is legal. For example, color is legal on all objects, but not on any properties. If there are any ambiguities in what tags may come where, it is up to you to discover them and request clarification.

OO Design

Keep in mind that this is a project using C++. As such, you are expected to design in a proper OO-fashion. As discussed in class, pay special attention to your class structure, and develop a logical hierarchy for all the objects in the simulation. Do also remember that as this is a C++ assignment, any usage of a C solution is not permitted if there is a C++ replacement, equivalent or workaround.

Coordinate System

For this assignment, the location (0,0) is the upper left corner of the simulation. The x-dimension grows to the right and the y-dimension grows to downward.

Assignment Parts:

a) (250 points) Create a program called checkconf that reads a configuration file. The program takes the filename as a command-line argument. The program validates the configuration file and then outputs to stdout a representation of the simulation environment, followed by a list of the objects and properties in the simulation, sorted by row then column.
For the representation, each square in the simulation is represented by four characters. Empty squares are represented by four underscores. If a square has an object in it, the first two characters for that square are replaced by that object's Display attribute. Similarly, if a square has a property, the last two characters are instead that property's Display attribute. If there is more than one property in a square, the last two characters should instead be XX. Each square is separated from the previous one by a single space, and newlines should be inserted between rows of squares.
For the list of objects and properties, display all contents of a square with the following information: xloc, yloc, Object Type, Object Name, Object Color, Property 1 Type, Property 1 Name, Property 2 Type, Property 2 Name, etc. Skip the Object information if there is no object present, and skip color if it isn't defined. If an item has no name, use Display as its name. Include energy for all objects that have an energy tag (for both energy contents and energy cost). See the example for further clarification.
If the configuration file is in any way invalid, output that error to stderr, and exit gracefully.

Sample output:

```
____ A1__ ____ ____
RO__ ____ BBFP ____
__MU ____ ____ __XX

Location: 0, 1
Type: Robot
Name: Steak Sauce
Energy Contents: 10576

Location: 1, 0
Type: Earth Rock
Name: RO

Location: 1, 2
Type: Ball
Name: Big Blue Bouncing Ball
Color: Orange
Type: Lava
Name: Flaming Lava Pit!!! Danger Will Robinson!!!
Energy Cost: 5

Location: 2, 0
Type: Mud
Name: Muddy McMudd

Location: 2, 3
Type: Water
Name: Fountain of Youth
Energy Cost: 1
Type: Fog
Name: Cloud 9
```

b) (150 points) Create a program called generateconf that prints a configuration file to stdout. The program, in its standard usage, takes as its first two arguments the width and height to create. Following those arguments are any number of the following flags:

-<Type> -l <x,y> //Chooses location to place instance of type
-<Type> -c <p> //Places p instances of Type at random locations

Each flag may be followed by any legal configuration file tag and its value.

For this usage, Types with spaces in them such as Energy Pill will have the spaces replaced with dashes, i.e. energy-pill, just as in the configuration file.

Example command line:

generateconf 20 25 -robot -l 5,6 -robot -l 16,17 -turncost 5 -energy-pill -c 37 -block -l 0, 0

will create two Robots at the specified locations, setting the turn cost of the second one, and a block at the specified location. It will also create 37 energy pills randomly throughout the sim.

For the Display of each object and property, if not entered, use the first character of the type and an index of how many instances of that object are in the sim. If there are more than 9 of a given object, restart from 1. For example, the 5^{th} energy pill added would be E5.

generateconf must also support the following special arguments:

- -threeblock
    - Create three red blocks and three blue blocks. The sim will be 30 width x 27 height. Create the three red blocks evenly spaced from each other and the from the walls in a vertical line 5 columns from the left, and the blue blocks 5 columns from the right. Create a blue robot in the lower left hand corner and a red robot in the lower right hand corner.
- -fiveitem
    - Create a sim the same as –threeblock, except that instead of three blocks place 5 items, alternating block ball block. The size of this sim is 30 width by 35 height.

It is in your interest to code these arguments in an extensible way, as it is very possible that by the final due date more built-in special arguments will be added.

If any of the arguments do not match correct formatting, print a usage statement and quit gracefully.

c) (350 points) Create a program called simulate that reads a configuration file and simulates the world according to the rules defined above. The simulation will have to create supporting objects to run the simulation. The simulation must communicate with the Robot Brains to determine their moves during the simulation. To do this, you will be provided with a program called piper that will be responsible for facilitating the communication.
To communicate with the Robots, piper will redirect standard input and output such that the Robots will be able to print their move to stdout and read in the info for the next move from stdin. On the simulate side, piper will create file descriptors that simulate will open, two for each Robot for input and output. The simulation takes the configuration file as its first command line argument and the number of turns to simulate as the second argument. The third argument is a string defining the mode of the program (mode is defined later).
After the first three command-line arguments, simulate takes 2 * n command line arguments as integers representing file descriptors, where n is the number of robots in the simulation. To use these file descriptors, the simulator must call fdopen, which returns a FILE * that can be used via regular C functions. For each set of these fd's, the first will be open for output to write to a robot, the second will be for input to read from a robot. Thus, if there were two robots in the sim, command-line arguments 4 and 5 would be for communicating with the first robot, and arguments 6 and 7 would be for communicating with the second robot. Use man fdopen for more instructions on using that function.

The robots and simulation will communicate using the following message formats:

1. Initialization
   At the start of the sim, the sim will send a single message to each Robot. This message will consist of:
   - Message Code
     - This will always be a 0 for the Initialization message.
   - initial energy
   - initial coordinates (xloc then yloc)
   - Robot color
   - sim size (height then width)
   - number of other robots in the sim.
2. Move Request
   Every turn including before the first move, the sim will send to each Robot a move request message. This message will consist of:
   - Message Code
     - This code will always be a 1 for the Move Request
   - Seen Item Information
     - This consists of a single number representing the number of things Seen by the robot, as defined above.
     - This number is followed by a list of the objects seen. Each object has a type, xloc, yloc, color and, if applicable, energy cost or contents.
   - Probed Item Information
     - If the robot probed last turn, the next information is a list of n distances returned by the probe, starting from the robot's left.
   - Energy
     - Every 5th move, the sim will add the Robot's current energy contents.

3. Termination Request
   When the simulation completes, the sim must signal to all robots to terminate. To do this, it sends a single Message Code of 2.
4. Move Decision
   Each turn that the robot receives a Move Request, it will respond by putting a Move Decision onto its stdout. This message will consist of:
   - Action Code
     - This is a number between 0 and 4.
     - 0 = Move Forward
     - 1 = Turn
     - 2 = Fire
     - 3 = Probe
     - 4 = Do Nothing
   - Parameter
     - If the Action code requires a parameter, the next item will be the parameter .
     - +90 or -90 for turn clockwise or counterclockwise, respectively
     - Shot Strength, for firing

- Beam width, for probing

For all of these messages, each item of information is simply placed onto and read from the correct stream/File object, surrounded by whitespace of any sort.

The Mode command-line argument has two options, -quiet and -verbose.
In –quiet mode, the simulation performs the entire simulation. At the end, it prints out the state of the simulation in the exact same manner as Part A of this assignment.
In –verbose mode, the simulation performs the simulation a single step at a time. After each step, it prints out the state of the simulation in the way the first part of Part A does, i.e. only the grid-based representation, not the list-based part. It waits for the user to press any key, then does the next step. At the end, it prints out the full Part A printout, just like -quiet.

Piper usage:
The Piper program will be provided to redirect stdin and stdout in order to provide the Simulation with the FD's that it needs, and to redirect stdin and stdout of each Robot to the sim. Piper requires the following command-line arguments:

1. The name (and path, if necessary) of the simulate executable.
2. The name (and path, if necessary) of each Robot Brain executable.
3. The last three arguments to Piper are the three arguments that simulate requires (config file, number of turns, and mode).

Sample usage:

piper ./simulate ./Robot1 ./Robot2 myConfigFile 100 –quiet.

Sample usage of simulate (note that you will never actually call this, piper will do it for you)

simulate myConfigFile 100 –quiet 150 100 175 10000

where the last 4 numbers are File Descriptors, opened by piper.

You will also be provided with a simple robot for testing that simply moves and turns in a non-useful manner.

d) (300 points) Create Robot Brains. Each robot brain has the following in common:
Each robot is a standalone executable that doesn't have any knowledge of the simulation. A robot reads from stdin, first the Initialization message and then each subsequent Move Request. It uses its algorithm and the information passed in to decided what to do, and outputs that Move Decision in the format defined above. When the Robot receives the Termination request, it exits.
If you feel there is additional information you need or would like from the simulation to write an intelligent algorithm, please ask; the message format is subject to change if it is determined that it should be extended (or narrowed).

You will create Robots for the following algorithms:

- Create a robot executable called PillAddict that searches the board for energy pills. It's sole objective is to gain as many energy pills as possible.
- Create a robot executable called Sentry that finds the edge of the grid and traverses the perimeter of the grid until it reaches its original spot, at which point it reverses and traverses the perimeter in the opposite direction.
- Create a robot executable called TriBot that determines the color of the rocks lining each of the left and right sides of the simulation. It then searches for all blocks of its own color and moves them to the side that is that color. The robots goal is to have all blocks of its color within 3 spaces of 'its' side of the board, effectively defining an endzone 3 cells deep. (This robot may assume it is placed on a board lined with rocks of its own color on one side, and another color on the other side).

e) (100 points) Write an appropriate makefile for all programs above. Also include a target called all (which should also be the default target) that creates everything. Also include a target called clean that removes all object files, and a target called cleandist that remove all generated files created.

f) (150 points) Create class diagram(s) to thoroughly document the design of your project. Preferably, these should be created digitally using a program like Astah Community, Visual Studio, or even something as simple as Google Docs. If you must, you may submit a scanned-in hand-drawn document, but ensure that the document is very clean and understandable. The submitted files should be .jpgs. Regardless of the image format, submit an English document that describes your design. It should include a breakdown of your class hierarchy, explain the purposes of classes, and general information about your design such as explaining why you broke the classes down the way you did.

Documentation:

Documentation and proper style will be at least 15% of the grade.