CSc 256 Chapter 4 Assignment
Tracking Two Particles

Due 4pm Thursday 10/13/2016
(Standard 48-hour grace period for 75% of credit)
(4% of your grade; 80/100 for correctness, 20/100 for documentation)
(This document is available from the CSc 256 iLearn site.)

For this project, you are given a C++ program that simulates the motion of two particles in a box. The boundaries of the box are defined on a x-y grid as 0<=x<=10, 0<=y<=10. The two particles (one red and one green) have some initial positions (set by the user) and initial velocities (hardcoded in the program). The particles will move around in the box. When a particle gets too close to a wall, it will bounce off the wall. When the two particles get too close to each other, they will collide and the simulation will end. You will translate this program faithfully, following all function call guidelines and MIPS register use conventions.

Each particle has an array which keeps track of its position and velocity. Elements 0 and 1 are the (x,y) coordinates of the position of the particle. Elements 2 and 3 are the (x,y) components of the velocity of the particle, representing the number of units the particle will move in either the x or y direction, in one cycle.

The C++ program, **particles.cc**, can be found on iLearn. An example simulation:

```
unixlab% ./a.out
Enter x-coordinate for red particle (0 to 10):0
Enter y-coordinate for red particle (0 to 10):8
Enter x-coordinate for green particle (0 to 10):8
Enter y-coordinate for green particle (0 to 10):10
cycle 0
red particle (x,y,xVel,yVel): 0 8 1 1
green particle (x,y,xVel,yVel): 8 10 -1 -1

cycle 1
red particle (x,y,xVel,yVel): 1 9 1 1
green particle (x,y,xVel,yVel): 7 9 -1 -1

cycle 2
red particle (x,y,xVel,yVel): 2 10 1 1
green particle (x,y,xVel,yVel): 6 8 -1 -1

cycle 3
red particle (x,y,xVel,yVel): 3 9 1 -1
green particle (x,y,xVel,yVel): 5 7 -1 -1

Collison: oops, end of simulation!
```

```
red particle (x,y,xVel,yVel): 4 8 1 -1
green particle (x,y,xVel,yVel): 4 6 -1 -1
```

The C++ program is fairly straightforward, with several nested function calls. The update() function has prototype

```
void update(int *arg)
```

arg is an array that contains the position and velocity of a particle. The update() function will update the parameters of the particle in one cycle.

The function findDistance() is called by both the main program and update(). It has prototype

```
int findDistance(int arg0, int arg1, int arg2, int arg3)
```

It returns the Manhattan distance between two particles, or a particle and a wall. The particles are at (arg0,arg1) and (arg2,arg3). If one of the coordinates is –1, it is understood that the corresponding particle is actually a wall, and findDistance() returns the distance between a particle and a wall.

A copy of the C++ program **particles.cc** can be found on iLearn. The file **particlesMain.s** contains the main program, already translated into MIPS assembly language. Your functions will go in a separate file. To run the main program and functions together, all you have to do is load them separately into spim. Suppose the main program is in p1main.s and the functions in p1.s:

```
(spim) load "p1main.s"
(spim) load "p1.s"
(spim) run
```

You must translate the update() function, and the findDistance() function. Write the functions exactly as described in this handout. Do not implement the program using other algorithms or tricks. Do not even switch the order of the arguments in function calls; you must follow the order specified in the C++ code. The purpose of this program is to test whether you understand nested functions. If you wish to make changes to the algorithm, you must first check with the instructor.

Your functions should be properly commented. Each function must have its own header block, including the prototype of the function, the locations of all arguments and return values, descriptions of the arguments and how they are passed, and a description of what the function does. Refer to the Programming Style handout from Chapter 2 slides and http://unixlab.sfsu.edu/~whsu/csc256/PROGS/4.2.s for guidelines on how to comment your code.

You should try to make your code efficient. For example, your loops should follow the efficient form presented in class. Points will be deducted for obvious inefficiencies.

Submission: submit your functions only, in a single file, using the iLearn submission link.

All your code (the two functions only, **not** the main program) should be in a single file. 80% of your grade is for correctness. 20% is for clarity/documentation.

Grading scheme:

The 20% for clarity/documentation is only assigned for *working* code. Hence, if you turn in beautiful comments for a program that crashes, you will not be awarded 20 d points for the comments.

If your program has non-trivial syntax errors, you will receive 10 or fewer points.

If your program has no syntax errors, but fails horribly (crashes, infinite loop etc), you will receive 30 or fewer points.

Partial credit depends on how well your code runs, efficiency, and adherence to register use conventions. Rough guidelines:

generation of correct results	50 points
efficient code (especially loop tests)	10 points
correct register use conventions	20 points

Correct register use conventions include having your arguments in \$a?, return values in \$v?, correct stack handling etc. Refer to the function specifications in this handout, and Chapter 4 slides.

Please consult me by email or via the iLearn forum, if you have trouble with syntax errors or run-time errors. As usual, it's ok to post <=5 lines of code, but not more than that.

```
//
//       CSc 256 Project 1 Two-Particle Simulation
//       by William Hsu
//       9/27/2011
//
//       There is a red particle and a green particle in a box.
//       The box has (x,y) coordinates 0 <= x <= 10, 0 <= y <= 10.
//       The user enters initial (x,y) coordinates for the two
//       particles. The velocity of a particle is given as
//       an (x,y) pair, where each coordinate is the distance
//       the particle moves in that direction, in one cycle.
//       Initially, the red particle starts with velocity (1,1),
//       and the green particle starts with velocity (-1,-1).
//
//       The program simulates the movement of the two particles.
//       When a particle reaches a wall, it will bounce
//       off the wall. When the two particles are too close together,
//       they will collide and the simulation will end.
//
//       Each particle is represented by an array of four ints.
//       Element 0 and 1 are the (x,y) coordinates of the particle.
//       Element 2 and 3 are the (x,y) velocity components of the
//       particle.

#include <iostream>
void updatePoint(int *);
int findDistance(int, int, int, int);

int main()
{
  int redData[4], greenData[4];
  int i,cycle=0, dist;

  // set initial conditions

  redData[2] = 1;
  redData[3] = 1;
  greenData[2] = -1;
  greenData[3] = -1;

  cout << "Enter x-coordinate for red particle (0 to 10):";
  cin >> redData[0];
  cout << "Enter y-coordinate for red particle (0 to 10):";
  cin >> redData[1];
  cout << "Enter x-coordinate for green particle (0 to 10):";
  cin >> greenData[0];
  cout << "Enter y-coordinate for green particle (0 to 10):";
  cin >> greenData[1];

  do {
    // display state of particles
    cout << "cycle " << cycle << endl;
    cout << "red particle (x,y,xVel,yVel): " << redData[0] << " " <<
redData[1]
         << " " << redData[2] << " " << redData[3] << endl;
    cout << "green particle (x,y,xVel,yVel): " << greenData[0] << " "
         << greenData[1] << " " << greenData[2] << " " << greenData[3]
         << endl << endl;

    // update particle positions and velocities
    updatePoint(redData);
    updatePoint(greenData);

    // check distance between particles
    dist = findDistance(redData[0], redData[1], greenData[0],
greenData[1]);
    cycle++;
  } while ((dist > 2) && (cycle < 10));

  if (dist <= 2) {
    cout << "Collison: oops, end of simulation!\n";
    cout << "red particle (x,y,xVel,yVel): " << redData[0] << " " <<
redData[1]
         << " " << redData[2] << " " << redData[3] << endl;
    cout << "green particle (x,y,xVel,yVel): " << greenData[0] << " "
         << greenData[1] << " " << greenData[2] << " " << greenData[3]
         << endl << endl;
  }

}

// update position and velocity of particle
// arg is array of data for particle

void updatePoint(int *arg)
{
  int distance;

  distance = findDistance(arg[0],arg[1],0,-1);
  if ((distance < 1) && (arg[2] < 0))
    arg[2] = - arg[2];
  distance = findDistance(arg[0],arg[1],10,-1);
  if ((distance < 1) && (arg[2] > 0))
    arg[2] = - arg[2];
  distance = findDistance(arg[0],arg[1],-1,0);
  if ((distance < 1) && (arg[3] < 0))
    arg[3] = - arg[3];
  distance = findDistance(arg[0],arg[1],-1,10);
  if ((distance < 1) && (arg[3] > 0))
    arg[3] = - arg[3];
  arg[0] = arg[0] + arg[2];
  arg[1] = arg[1] + arg[3];
  return;
}

// find Manhattan distance between two particles, or between
// a particle and a wall
// (arg0,arg1) are the (x,y) coordinates for first particle/wall
// (arg2,arg3) are the (x,y) coordinates for second particle/wall

int findDistance(int arg0, int arg1, int arg2, int arg3)
{
  int distX, distY;
  distX = arg0 - arg2;
  if (distX < 0)
    distX = - distX;
  distY = arg1 - arg3;
  if (distY < 0)
    distY = - distY;
  if ((arg0 < 0)  || (arg2 < 0))
    return distY;
  else if ((arg1 < 0)  || (arg3 < 0))
    return distX;
  else
    return distX + distY;
}
```