CSE 241 Programming Assignment 5

DUE

May 12, 2023, 23:55

Description

In this PA, you are going to use the code you have written in PA7. Instead of `ant` and `doodlebug`, you are going to have robots.

We have four different types of robots: `optimusprime`, `robocop`, `roomba`, and `bulldozer`. To represent one of these robots we might define a `Robot` class as follows:

Some of the members are given the others are left to you so that you can decide. Decide which of the members are going to be `private` or `public`.

```
class Robot
{

    //a member data which defines the type
    //a member data which stores the strength
    //a member data which stores the hitpoints
    //a helper function which returns the robot type
    Robot( );
    Robot(int newType, int newStrength, int newHit, string name);
    // Initialize robot to new type, strength, hit points
    // Also add appropriate accessor and mutator functions
    // for type, strength, and hit points
    int getDamage();
    // Returns amount of damage this robot
    // inflicts in one round of combat
};
```

Here is an implementation of the `getType( )` function: We are not going to use this function. Instead we will define this function as `virtual` and provide different implementations for different type of robots.

```
string Robot::getType()
{
    switch (type)
    {
        case 0: return "optimusprime";
        case 1: return "robocop";
        case 2: return "roomba";
        case 3: return "bulldozer";
    }
    return "unknown";
}
```

The `getDamage( )` function outputs and returns the damage this robot can inflict in one round of combat. The rules for calculating the damage are as follows:

- Every robot inflicts damage that is a random number `r`, where `0 < r <= strength`.
- `humanic` robots have a `10%` chance of inflicting a tactical nuke attack which is an additional `50` damage points. `optimusprime` and `robocop` are `humanic`.
- With a `15%` chance `optimusprime` robots inflict a strong attack that doubles the normal amount of damage.
- `roomba` robots are very fast, so they get to attack twice.

A skeleton of `getDamage( )` is given below:

```
int Robot::getDamage()
{
```

```
    int damage;
    // All robots inflict damage which is a
    // random number up to their strength
    damage = (rand() % strength) + 1;
    cout << getType() << " attacks for " <<
    damage << " points!" << endl;
   //calculate additional damage here depending on the type


   //
    return damage;
}
```

One problem with this implementation is that it is unwieldy to add new robots. Rewrite the class to use inheritance, which will eliminate the need for the variable type. The `Robot` class should be the base class. The classes `bulldozer`, `roomba`, and `humanic` should be derived from `Robot`. The classes `optimusprime` and `robocop` should be derived from `humanic`. You will need to rewrite the `getType( )` and `getDamage( )` functions so they are appropriate for each class. For example, the `getDamage( )` function in each class should only compute the damage appropriate for that object. The total damage is then calculated by combining the results of `getDamage( )` at each level of the inheritance hierarchy. As an example, invoking `getDamage( )` for a `optimusprime` object should invoke `getDamage( )` for the `humanic` object which should invoke `getDamage( )` for the `Robot` object. This will compute the basic damage that all robots inflict, followed by the random `10%` damage that `humanic` robots inflict, followed by the double damage that `optimusprime` inflict. Also include mutator and accessor functions for the private variables.

Setup

We are going to have a grid just like the application in PS7. Then we create robots and randomly place them in the cells of the grid.

- `grid_size`: 10x10
- `initial_count_of_each_robot_type`: 5

Create names for each robot according to the following format:

- `name`: `<type_name_of_the_robot>_<creation_sequence_number_for_each_type>`
- example: `robocop_0`
- `creation_sequence_number_for_each_type` starts from `0` and incremented.
- so, initially you will have `robocop_0, robocop_1,...robocop_5`, `bulldozer_0, bulldozer_1...bulldozer_5`, etc...

Initial values for each robot type is as follows:

- `optimusprime`: strength:100, hitpoints:100
- `robocop`: strength:30, hitpoints:40
- `roomba` strength:3, hitpoints:10
- `bulldozer` strength:50, hitpoints:200

Simulation

Repeat until only one of the robots survive:

- Visit every cell of the grid. If the cell is occupied by robot `R`:
  - `R` tries to move up, down, left or right.
  - If the movement direction is occupied by another robot, `R` fights with that robot until one of them is dead. (`the fight loop`)
  - **If the cell is empty, `R` moves to that location and keeps moving until it hits another robot.**
  - Every robot has one chance of fight for every step of simulation just like in the `ant` and `doodlebug` simulation. (you have to keep a flag in every robot and skip the robot if it is already moved).

the fight loop

Lets say, robot R(attacker) tries to fight with robot S(victim). Here is the algorithm:

Repeat until R or S dies:

- `R` calls `getDamage()`. `getDamage()` returns `d_r`.
- `hitpoints` of `S` is decremented by `d_r`.
- print `hit_message`(see `hit_message` for details)
- If `S` is dead, return.
- `S` calls `getDamage()`. `getDamage()` returns `d_s`.
- `hitpoints` of `R` is decremented by `d_s`.
- print `hit_message`(see `hit_message` for details)

`hit_message`

The hit message has two lines. The format of hit message is as follows:

- `<name_attacker>(<hitpoits_attacker>) hits <name_of_the_victim>(<hitpoints_victim_before_hit>) with <damage_inflicted>`
- `The new hitpoints of <name_victim> is <hitpoins_victim>`
- Example:
- `roomba_1(10) hits robocop_4(10) with 3`
- `The new hitpoints of robocop_4 is 7`

Death of a robot

If the `hitpoints` is less than or equal to `0`, the robot is announced as dead. Dead robots should be removed from the grid.

Turn In

- A zip file containing all the `.cpp` and `.h` files of your implementation. Properly name your files according to the classes you your. Put your driver program(main function) in `main.cpp`.
- Create a simple `MAKEFILE` for your submission. (You can find tutorials for creating a simple make file. If you are having difficulty, send me an email.)
- Name of the file should be in this format: `<full_name>_PA5.zip`. Don't send `.rar` or `.7z` or any other format. Properly create a `.zip` file from your source files.
- You don't need to use an IDE for this assignment. Your code will be compiled and run in a command window.
- Your code will be compiled and tested on a Linux machine(Ubuntu). GCC will be used.
- Make sure you don't get compile errors when you issue this command : `g++ -std=c++11 <any_of_your_files>.cpp`.
- Makes sure you don't get link errors.

Late Submission

- Not accepted

Grading (Tentative)

- `Max Grade` : `100`.
- Multiple tests will be performed.

All of the followings are possible deductions from `Max Grade`.

- Do **NOT** use hard-coded values. If you use you will loose `10pts`.
- No submission: `-100`. (be consistent in doing this and your overall grade will converge to `N/A`) (To be specific: if you miss `3` assignments you'll get `N/A`)
- Compile errors: `-100`.
- Irrelevant code: `-100`.
- Major parts are missing: `-100`.
- Unnecessarily long code: `-30`.
- Inefficient implementation: `-20`.
- Using language elements and libraries which are not allowed: `-100`.
- Not caring about the structure and efficiency: `-30`. (avoid using hard-coded values, avoid hard-to-follow expressions, avoid code repetition, avoid unnecessary loops).
- Significant number of compiler warnings: `-10`.
- Not commented enough: `-10`. (Comments are in English. Turkish comments are not accepted).
- Source code encoding is not `UTF-8` and characters are not properly displayed: `-5`. (You can use 'Visual Studio Code', 'Sublime Text', 'Atom' etc... Check the character encoding of your text editor and set it to `UTF-8`).
- **Not using virtual functions `-100`.**
- **Not using class inheritance `-100`.**
- **Not de-allocating dynamic memory `-20`.**
- Output format is wrong `-20`.
- Calculation is wrong `-20`.
- Infinite loop: `Fails the test`.
- Segmentation fault: `Fails the test`.
- Fails 5 or more random tests: `-100`.
- Fails the test: `deduction up to 20`.
- Unwanted chars and spaces in output: `-30`.
- Submission includes files other than the expected: `-10`.
- Submission does not follow the file naming convention: `-10`.
- Sharing or inheriting code: `-200`.