Assignment 2: Roguelike

Due: Wednesday, October 11, 2017 at 4:20pm
Submission cutoff: Saturday, October 14, 2017 at 4:20pm

Overview

As mentioned in lecture, Lua doesn't come with a built-in object system — so we can make our own! In this assignment, you'll build a homebrew class system and use it to implement a simple text-based adventure game. Then you will use the Lua C API to implement an accelerated version of one of the classes in the game.

Setup

On the Rice machines:

```
cp -r /afs/ir/class/cs242/assignments/assign2 assign2
```

On your local machine:

```
scp -r <SUNetID>@rice.stanford.edu:/afs/ir/class/cs242/assignments/assign2 assign2
```

Submitting

You must implement functions in:

- `class.lua`
- `monster.lua`
- `native_point.c`

Then upload the files in your copy of `assign2` to Rice. To submit, navigate to the assignment root and execute:

```
python /afs/ir/class/cs242/bin/submit.py 2
```

In order to verify that your assignment was submitted, execute

```
ls /afs/ir/class/cs242/submissions/assign2/<SUnet ID>
```

You should see the timestamps of your submissions.

Part 1: Class system

In this section of the assignment, you will implement a library that provides a single-inheritance class system with **public methods** and **private data members**, similar to a restricted subset of Java or C++. Here's a simple example:

```lua
local class = require("class")
local Counter = class.class(
  class.Object, {
    constructor = function(self, n)
      self.n = n
    end,
    data = {
      counter = 0,
    },

    methods = {
      incr = function(self)
        self.counter = self.counter + self.n
      end,

      value = function(self)
        return self.counter
      end
    }
  })

local inst = Counter.new(5)
inst:incr() -- can call methods from the outsdie
inst:incr()
assert(inst:value() == 10)
assert(inst.counter == nil) -- can't access data members
```

We can extend the above example to demonstrate inheritance:

```
local WeightedCounter = class.class(
  Counter, {
    constructor = function(self, n, weight)
      Counter.constructor(self, n)
      self.weight = weight
    end,

    methods = {
      incr = function(self)
        self.counter = self.counter + self.n * self.weight
      end,

      set_weight = function(self, weight)
        self.weight = weight
      end
    }
  })

local inst = WeightedCounter.new(5, 0.2)
inst:incr()               -- overriden in WeightedCounter
assert(inst:value() == 1) -- inherited from Counter
inst:set_weight(2.0)      -- new function in WeightedCounter
inst:incr()
assert(inst:value() == 11)
assert(inst:isinstance(WeightedCounter)) -- isinstance checks the class of inst
assert(inst:isinstance(Counter))         -- a child is an instance of the parent
```

Specification

The core of this section will be implementing the `class` function in `class.lua`. That function fulfills the following spec:

- `class` takes two arguments, a parent class `parent` (either `Object` or a class returned by `class`) and a description of a child class `child` with the following structure:

  ```
  {
    constructor = function(self, ...)
      -- to be called when an instance is created
    end,

    data = {
      -- table of initial values for private members
      -- e.g. x = 3
    },

    methods = {
      -- table of public methods that can be called on an instance
  ```

```
      -- e.g. foo = function(self) print(self.x) end
    },

    metamethods = {
      -- table of metamethods assigned to each instance
      -- e.g. __index = function(t, k) return nil end
    }
  }
```

- `class` returns a `Class` table that fulfills the following specification:
  - **Constructors:** `Class.new(...)` is a function that creates a new instance `t` and runs either the child's `constructor(t, ...)` if it is defined, and the parent's constructor otherwise.
  - **Methods:** An instance `t` can have any of the methods in its class's `methods` table or any of its parent classes' `methods` tables invoked on the instance. All methods take the `t` (or `self`) as the first parameter.
  - **Data:** When inside a method, the `self` variable should provide access to both private data members as well as public methods. Any data members should be initialized to their value in the child or any parent's `data` table. Any new parameters added to `self` , e.g. in the constructor, must also be private (i.e. not accessible outside the class methods).
  - **Metamethods:** An instance `t` should have any metamethods in the child and any inherited `metamethods` set to its metatable, with the exception of the `__index` metamethod.
  - **isinstance**: `t` has an additional method `t:isinstance(Class)` that returns true if `t` is an instance of the given `Class` or a subclass of `Class` .

Testing

We have provided you a small set of basic tests. You can run them similarly as before:

```
lua class_tests.lua
```

Getting started

This is expected to be the hardest part of the assignment, and will take some time to think through. To help you along the way, we've tried to break the problem down into sub-parts for you to tackle individually.

1. Methods

```lua
local Part1Class = class(
  Object, {
    methods = {
      setX = function(self, x)
        self.x = x
      end,

      getX = function(self)
        return self.x
      end
    }
  })

local inst = Part1Class.new()
inst:setX(10)
assert(inst:getX() == 10)
```

Here, use the metatable approach discussed in class to associate the methods in `Part1Class` with the instance `inst` . Don't worry about whether `x` is private or public for now.

2. Data

```lua
local Part2Class = class(
  Object, {
    data = {
      x = 0
    },
    methods = {
      setX = function(self, x)
        self.x = x
      end,

      getX = function(self)
        return self.x
      end
    }
  })

local inst = Part2Class.new()
assert(inst:getX() == 0)
assert(inst.x == nil)
inst:setX(10)
assert(inst:getX() == 10)
assert(inst.x == nil)
```

To get private data members, essentially you will need two instances: a private instance that contains the data members (`x` in this case) and a public instance that just contains the class methods. Here `inst` would be the public instance. Normally calling `inst:getX()` desugars to `inst.getX(inst)` which calls the public method using the public instance. However, consider using closures to force methods to be called using the private instance. For example:

```
local public_inst = {}
local private_inst = {x = 0}
public_inst.getX = function(self, ...)
  return Part2Class.methods.getX(private_inst, ...)
end
```

You will want to use the `__index` metamethod to enable the `private_inst` to refer to methods on the `public_inst`.

Note: having default values for data gets a little tricky when dealing with tables. For example:

```
local Class = {x = {y = 1}}
local inst = {}
setmetatable(inst, {__index = Class})
inst.x.y = 2
print(Class.x.y) -- it's now 2 as well. We just modified the Class!
```

You may assume that a class instance will never do any such kinds of mutations that would affect the data table within the Class table. Furthermore, you can assume that method implementations will never try to return private instances, or otherwise cause private instances to escape the current function call.

3. Constructors

```
local Part3Class = class(
  Object, {
    constructor = function(self, x)
      self.x = x
    end,

    methods = {
      getX = function(self)
        return self.x
      end
    }
  })

local inst = Part3Class.new(5)
assert(inst:getX() == 5)
assert(inst.x == nil)
```

Constructors should be invoked during `Class.new`.

4. Metamethods

```lua
local Part4Class = class(
  Object, {
    metamethods = {
      __add = function(a, b) return 0 end
    }
  })

local inst = Part4Class.new()
assert(inst + 0 == 0)
```

Metamethods need to be assigned to each instance generated by `new`. Implementations of metamethods will only use methods on objects, and will not try to directly access object properties. Futhermore, no class definition will try to override the `__index` metamethod, thus letting you use the `__index` metamethod for method and data lookup.

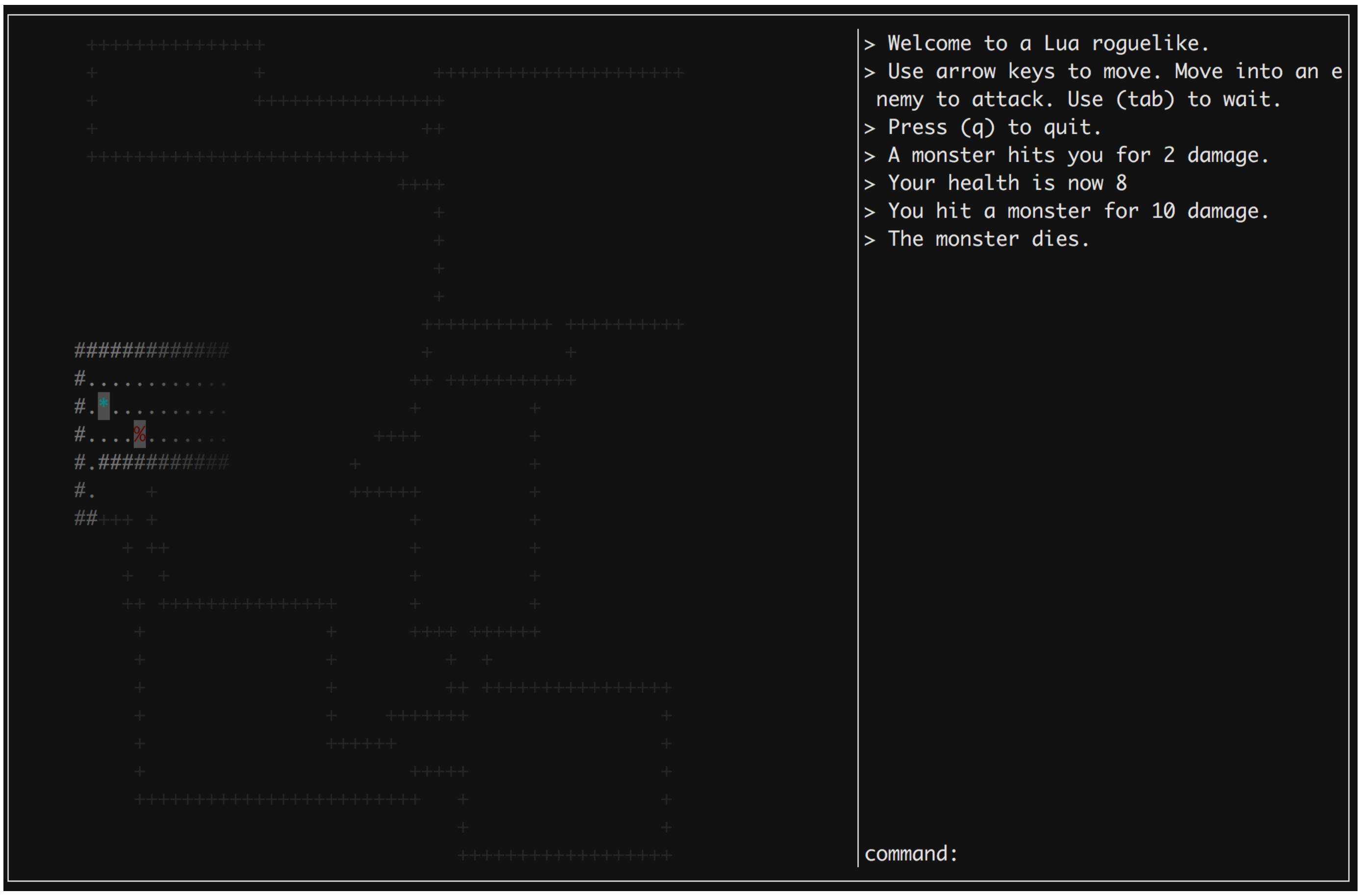
5. Inheritance

```lua
local Part5BaseClass = class(
  Object, {
    data = {
      x = 0,
      y = 1
    },

    methods = {
      getX = function(self)
        return self.x
      end,

      setX = function(self, x)
        self.x = x
      end
    }
  })

local Part5ChildClass = class(
  Part5BaseClass, {
    data = {
      x = 2
    },

    methods = {
      getY = function(self)
        return self.y
      end,

      setX = function(self, x)
        self.x = x * 2
      end
    }
  })

local inst = Part5ChildClass.new()
assert(inst:getY() == 1)
assert(inst:getX() == 2)
inst:setX(10)
assert(inst:getX() == 20)
assert(inst:isinstance(Part5ChildClass))
assert(inst:isinstance(Part5BaseClass))
assert(inst:isinstance(Object))
```

Inheritance for methods and data members should use a similar mechanism, i.e. metatables, to define fallbacks. An instance should fallback to its class, its class should fallback to its parent, and so on.

Note that although a similar mechanism could theoretically work for metamethods, in practice Lua doesn't allow metatables to themselves have metatables, so you will need to copy all the metamethods from the child and each of its parents into a single metatable.

Part 2: Roguelike



```
> Welcome to a Lua roguelike.
> Use arrow keys to move. Move into an e
 nemy to attack. Use (tab) to wait.
> Press (q) to quit.
> A monster hits you for 2 damage.
> Your health is now 8
> You hit a monster for 10 damage.
> The monster dies.

command:
```

To provide more thorough testing for your class library and provide you a chance to use it in practice, we have built most of a roguelike text-based game for you. This game consists of wandering around a dungeon and running into monsters until they are vanquished. We have distributed a reference solution for the game so you can try it out:

```
lua solution.bin
```

If your class library is fully functional, then the game should work almost as expected. To run the game with your version of the class library, run:

```
lua roguelike.lua
```

The game is roughly structured as follows: the map is a 2D grid of tiles, represented by integer coordinates described with the `Point` class. The grid contains instances of the `Entity` class (see `entity.lua`), which represent movable objects including the user (the hero) and the three monsters. Entities are built around callbacks—for example, if an entity collides with another entity, then the game will call the `Collide` method. On every game tick (whenever the user inputs a move command), the game will call `Think` on every entity.

Monsters

Your task for this portion is to implement one of the characters in the game, the monster that runs towards the player to attack. You will implement the `Think` method in the `Monster` class in `monster.lua`.

The monster's `Think` should follow this specification: if the monster can see the hero, then it attempts to move one step in the direction of the hero. That's it! The solution should be short (5-6 lines of code), and is primarily just to demonstrate an interaction with the class system.

To accomplish some of these goals (like visibility testing), you should look at the methods available on the `Entity` class in `entity.lua`, of which `Monster` is a subclass. Additionally, look at the `Hero` and `TryMove` methods on the `Game` class in `game.lua`. Note that you must use `TryMove` instead of `SetPos` when moving in order for the game to correctly handle collision detection.

Part 3: Fast points

One of the classes we use in the game is the `Point` class defined in `point.lua` that represents a 2D coordinate or vector. The class itself is relatively simple, but it's important that it be fast–although our game is relatively small, games frequently require millions of instances of classes like `Point`. In this final portion of the assignment, you will use the Lua C API discussed in lecture to create a C implementation of the `Point` class.

All the code for this portion is in `native_point.c`. Specifically, you will implement the following functions (prefixed with `point_`): `add`, `dist`, `eq`, `sub`, `x`, `y`, `setx`, `sety`, `tostring`. The end result will be a class that functions identically to the `Point` class in `point.lua`, except must faster! This native Point class does not need to implement `p:isinstance(cls)`.

> Note: we have already implemented the function `luaopen_native_point` which is the function called when you `require` the C library. This sets up the `Point` class table for you—you just need to implement the class's methods. Note that this function also creates the `"point_native"` metatable.

We have defined for you at the top a `point_t` struct that represents an instance of a `Point` with an `x` and `y` of type `lua_Number` (which will be `double` on our platforms). You will need pass around handles to `point_t` pointers using Lua's concept of *userdata*, or opaque C pointers. Specifically, you will need to investigate the following functions:

- `lua_newuserdata`
- `luaL_checkudata`
- `luaL_getmetatable`
- `lua_setmetatable`
- `lua_pushnumber`
- `lua_pushboolean`
- `lua_pushstring`

Note that userdata can have metatables just like normal tables, including the `__index` method. We have created for you the `"point_native"` metatable that contains an `__index` equal to the method list defined in `luaopen_native_point`, so setting a userdata's metatable to `"point_native"` will enable method lookup to work as expected.

Building and testing

To build a Lua module out of your C file, we use `gcc` to create a shared library. This build process is defined by the `Makefile`, so you can build by running:

```
make
```

To test your `native_point.c`, you can run a small benchmark script we have written:

```
lua benchmark_point.lua
```

This will both check that your native Point class works correctly and demonstrate its speedups relative to the equivalent Lua implementation. Your C implementation should be much faster! You can also test your implementation inside the game by running it with:

```
lua roguelike.lua --native-point
```

Getting started

I would start by implementing the `point_new`, `point_x`, and `point_setx` functions. Your goal for `new` is to create a new userdata representing the `point_t`. `point_x` should exercise returning values out of a C function into Lua, and `point_setx` demonstrates reading arguments from Lua into C.

Then implement/test the `point_add` function to make sure that you properly set the metatable on any points you created. After that, implementing the remaining functions should be relatively straightforward.