CISS245: Advanced Programming
Assignment 5

Name: _____

Objectives.

1. The purpose of this assignment is build a simple library for a `struct`.
2. Declare `struct` variable
3. Access member variables of a `struct` variable
4. Write function with `struct` parameters
5. Write function with `struct` return value

Follow the usual way of naming your folders. For question Q1, create a folder `a05q01` and put all `*.cpp` and `*.h` files into this folder. Etc. All folders should be kept in folder `a05` and `a05` should be in `a` in `ciss245`.

Although there are many questions, this is actually not a difficult assignment since this assignment is mainly a practice on working with `struct`.

We will write a simple simulation of several robots wandering about in a finite 2-dimensional world.

Each robot has the following:

1. name
2. energylevel
3. x–coordinate
4. y–coordinate

Here's our world with 3 robots

```
time: 50
   0123456789
  XXXXXXXXXXX
0 X          X
1 X  c    G  X
2 X     P    X
3 X P        X
4 X   XXXX   X
5 X   r      X
6 X   X      X
7 X   X   P  X
8 X d X      X
9 X   X      X
  XXXXXXXXXXX
<Robot: name=c: x=2, y=1, energylevel=90>
<Robot: name=d: x=3, y=5, energylevel=45>
<Robot: name=r: x=1, y=8, energylevel=75>
<PowerStation: x=1, y=3, energylevel=50>
<PowerStation: x=5, y=2, energylevel=50>
<PowerStation: x=7, y=7, energylevel=50>
<Gold: x=7, y=1>
```

The world is 10-by-10. Robots can only walk in the world. There are two walls of length 4 in the world (look for the `X` in the above output).

When you run the program, the two walls are erected randomly in the world (or "room" if you wish to think of it that way.) The walls are built randomly; they are either vertical or horizontal and of length 4.

Furthermore, there are power stations `P` in the world. There are 3 power stations and their positions are randomly selected. Their positions are chosen on available spaces, i.e., they are not in the walls and are not on top of each other and are not on the same position as the pot of gold (see below). Initially the power stations have a power level of 50. Power stations do not move; they are stationary.

There is also a pot of gold, `G`, placed randomly in the world. Of course the pot of gold cannot be placed in the wall! Also, the pot of gold cannot occupy the the same

position as a power station. `G` is also stationary.

The robots are initially randomly placed in the world all starting with energy level of 100.

When the simulation starts, the time is set to 100. For each iteration of the simulation, the time decrements by 1. The robots walk randomly and with each step, their energy level decrements by 1. Of course the robot cannot walk into a wall nor can two robots stand on the same position. Likewise robots cannot stand on the same position as the `P`'s or `G`. Here's the first few iterations of a simulation:

```
42
time: 100
   0123456789
  XXXXXXXXXXX
0 X          X
1 X  c    G  X
2 X     P    X
3 X P        X
4 X   XXXX   X
5 X   r      X
6 X   X      X
7 X   X   P  X
8 X d X      X
9 X   X      X
  XXXXXXXXXXX
<Robot: name:c, x=2, y=1, energylevel=100>
<Robot: name:d, x=1, y=8, energylevel=100>
<Robot: name:r: x=3, y=5, energylevel=100>
<PowerStation: x=1, y=3, energylevel=50>
<PowerStation: x=5, y=2, energylevel=50>
<PowerStation: x=7, y=7, energylevel=50>
<Gold: x=7, y=1>

time: 99
   0123456789
  XXXXXXXXXXX
0 X          X
1 X   c   G  X
2 X     P    X
3 X P        X
4 X   XXXX   X
5 X  r       X
6 X   X      X
7 X d X   P  X
8 X   X      X
9 X   X      X
  XXXXXXXXXXX
<Robot: name=c, x=3, y=1, energylevel=99>
<Robot: name=d, x=2, y=5, energylevel=99>
<Robot: name=r, x=1, y=7, energylevel=99>
<PowerStation: x=1, y=3, energylevel=50>
<PowerStation: x=5, y=2, energylevel=50>
<PowerStation: x=7, y=7, energylevel=50>
<Gold: x=7, y=1>
```

(The user entered 42 as the seed for the random generator. In the following, I will not display the user entry of the seed.)

When a robot moves to a position that is just next to a P, up to 10 points of energy

is transferred from the power station to the robot. Of course if the power station has a power level of 0, no power is transferred to the robot. When I say "next to" I mean the robot is one square to the left or right or top or bottom of `P`. Note that the energy is transferred once the robot moves to a square just next to the power station. Here's an example where Robot `c` is just next to a power station and will immediately get up to 10 points of energy:

```
   0123456789
  XXXXXXXXXXX
0 X          X
1 X       G  X
2 X    cP    X
3 X P        X
4 X   XXXX   X
5 X   r      X
6 X   X      X
7 X   X   P  X
8 X d X      X
9 X   X      X
  XXXXXXXXXXX
```

However in the following, `c` is not considered next to a power station:

```
   0123456789
  XXXXXXXXXXX
0 X          X
1 X    c  G  X
2 X     P    X
3 X P        X
4 X   XXXX   X
5 X   r      X
6 X   X      X
7 X   X   P  X
8 X d X      X
9 X   X      X
  XXXXXXXXXXX
```

Note that in each iteration of the simulation, two robots arrive at the power station at the same time. For instance at time 25 we can have:

```
   0123456789
  XXXXXXXXXXXX
0 X          X
1 X    c  G  X
2 X     P    X
3 X P    r   X
4 X   XXXX   X
5 X          X
6 X   X      X
7 X   X   P  X
8 X d X      X
9 X   X      X
  XXXXXXXXXXXX
```

and at time 24 we have this:

```
   0123456789
  XXXXXXXXXXXX
0 X          X
1 X       G  X
2 X    cP    X
3 X P   r    X
4 X   XXXX   X
5 X          X
6 X   X      X
7 X   X   P  X
8 X d X      X
9 X   X      X
  XXXXXXXXXXXX
```

Both `c` and `r` are next to the same power station. In this case `c` and `r` will get the same amount of energy points. For instance if the power station have 9 energy points, then they each get 4 (and then move away in the next iteration) leaving the power station with 1 energy point.

When a robot's energy level goes to 0, the robot stops moving.

The simulation stops when the time reaches 0 or when one of the robots found the gold, i.e., is next to `G`.

(Note that since randomization is involved, you will need to use the `rand()` function.)

Skeleton code for Robot

Th skelton code is provided only as a guide. It's of course incomplete and might contain errors.

You will provide useful functionalities for `Robot` structure variables.

Here's the header file for `Robot.h`:

```
/*****************************************************************

File  : Robot.h
Author: smaug

void init(Robot &, char name, int x, int y, int energylevel);
Sets the name, x, y, and energylevel to the arguments passed into the
function.

void print(const Robot & robot);
Prints the instance variables of the robot. If the name, x, y,
energylevel of robot is 'c', 10, 20, 30, then the output is
<Robot: name=c, x=10, y=20, energylevel=30>
A newline is printed at the end.

void move_north(Robot & robot);
Decrement the y value of robot and decrements the energylevel by 1.

void move_south(Robot & robot);
Increments the y value of robot and decrements the energylevel by 1.

void move_east(Robot & robot);
Increments the x value of robot and decrements the energylevel by 1.

void move_west(Robot & robot);
Decrements the x value of robot and decrements the energylevel by 1.

void inc_energylevel(Robot & robot, int d);
Increments the energylevel of robot by d.

void dec_energylevel(Robot & robot, int d);
Decrements the energylevel of robot by d.

*****************************************************************/

#ifndef ROBOT_H
```

```
#define ROBOT_H

struct Robot
{
    char name;
    int x, y;
    int energylevel;
};

void init(Robot &, char name, int x, int y, int energylevel);
void print(const Robot & robot);
void move_north(Robot & robot);
void move_south(Robot & robot);
void move_east(Robot & robot);
void move_west(Robot & robot);
void inc_energylevel(Robot & robot, int d);
void dec_energylevel(Robot & robot, int d);

#endif
```

```
/*************************************************************************

File  : Robot.cpp
Author: smaug

Implementation of functions prototypes in Robot.h.

*************************************************************************/
#include <iostream>
#include "Robot.h"

void init(Robot & r, char name, int x, int y, int energylevel)
{
    r.name = name;
    r.x = x;
    r.y = y;
    r.energylevel = energylevel;
}
```

Q1. The goal for Q1 is to implement the function prototype

```
void print(const Robot & robot);
```

of `Robot.h`. Of course the implementation is placed in `Robot.cpp`. Note that I have already given you the implementation of

```
void init(Robot &, char name, int x, int y, int energylevel);
```

(See `Robot.cpp`.)

To test the `print` function, here's the code for `main.cpp`.

```
#include <iostream>
#include <cstdlib>
#include "Robot.h"

void test_print()
{
    char name;
    int x, y;
    int energylevel;
    std::cin >> name >> x >> y >> energylevel;
    Robot r;
    init(r, name, x, y, energylevel);
    print(r);
    return;
}


int main()
{
    int seed;
    std::cin >> seed;
    srand(seed);

    int option = 0;
    std::cin >> option;
    switch (option)
    {
        case 1:
            test_print();
            break;
    }
    return 0;
```

```
}
```

So in the folder for this question, there are three files: `main.cpp`, `Robot.h`, and `Robot.cpp`.

Test 1

```
1 c 1 2 100
<Robot: name=c, x=1, y=2, energylevel=100>
```

Test 2

```
1 d 0 5 42
<Robot: name=d, x=0, y=5, energylevel=42>
```

Perform as many test cases as you need.

Q2. Now provide an implementation of the function

```
void move_north(Robot & robot);
```

function. (Of course you put this is `Robot.cpp`.)

```
#include <iostream>
#include <cstdlib>
#include "Robot.h"

void test_print()
{
    char name;
    int x, y;
    int energylevel;
    std::cin >> name >> x >> y >> energylevel;
    Robot r;
    init(r, name, x, y, energylevel);
    print(r);
    return;
}


void test_move_north()
{
    char name;
    int x, y;
    int energylevel;
    std::cin >> name >> x >> y >> energylevel;
    Robot r;
    init(r, name, x, y, energylevel);
    move_north(r);
    print(r);
    return;
}


int main()
{
    int seed;
    std::cin >> seed;
    srand(seed);

    int option = 0;
    std::cin >> option;
```

```
    switch (option)
    {
        case 1:
            test_print();
            break;
        case 2:
            test_move_north();
            break;
    }
    return 0;
}
```

The test case below will tell you what you need to do.

Test 1

```
2 c 5 3 100
<Robot: name=c, x=5, y=2, energylevel=99>
```

Note that the energylevel drops by 1 after the Robot moves by 1 step. You need *not* check that the Robot moves outside the world, i.e., you need not check that x and y are in the 0..9 range. However you need to check that the energy level is positive before allowing the Robot to move. For instance if the energylevel is 0, calling `move_north` will not change the position of the Robot.

Perform as many test cases as you need.

Q3. Now complete the

```
void move_south(Robot & robot);
```

function.

Note that all test code from Q1 and Q2 must be retained. In other words, Q3 builds on top of Q2. The test option here is 3. From Q2, you should be able to tell what the `move_south` function should achieve.

Q4. Now complete the

```
void move_east(Robot & robot);
```

function.

Note that all test code from Q1, Q2, and Q3 must be retained. In other words, Q4 builds on top of Q3. The test option here is 4. This note is similar for the rest of the assignment. So I will not be saying this again.

Q5. Now complete the implementation of

```
void move_west(Robot & robot);
```

function.

Q6. Now complete the implementation of

```
void inc_energylevel(Robot & robot, int d);
```

function.

The test function prompts for data for building the Robot and an integer value that will be used for incrementing the Robot's energy level. It then initializing the Robot with the value entered by the user and then call the `inc_energylevel` function. Finally it prints the Robot. Here's the test function. Of course it must be added to `main()`.

```
void test_inc_energylevel()
{
    char name;
    int x, y;
    int energylevel;
    std::cin >> name >> x >> y >> energylevel;
    Robot r;
    init(r, name, x, y, energylevel);
    int i;
    std::cin >> i;
    inc_energylevel(r, i);
    print(r);
    return;
}
```

Test 1

```
6 ? 7 1 100 5
<Robot: name=?, x=7, y=1, energylevel=105>
```

Q7. Now complete the

```
void dec_energylevel(Robot & robot, int d);
```

function.

Test 1

```
7 c 1 2 100 5
<Robot: name=c, x=1, y=2, energylevel=95>
```

Skeleton code for world

Although we do have all the positions of the robots in the robots themselves, it is still convenient to have all the locations marks in some kind of a "map". I'll call this array variable `world`. For instance it's useful for drawing purposes. We'll use a 2-dimensional array. a05q07/skel/world.cpp

```
/*********************************************************************

File  : world.cpp
Author:

Implementation of world.h.

*********************************************************************/
#include <iostream>
#include "world.h"

void init(char world[10][10])
{
}

// Other functions ...
```

a05q07/skel/world.h

```
/*********************************************************************
File  : world.h
Author:

void init(char world[10][10]);
Set all elements in the array to ' ' and erects the walls.

void print(char world[10][10]);
Prints the 2-d array world. This includes a border of 'X' characters
and integers 0-9 marking the x-coordinates going left to right and
y-coordinates going top to bottom. Here is an example.
   0123456789
  XXXXXXXXXXX
0 X          X
1 X   c   G  X
2 X     P    X
3 X P        X
4 X   XXXX   X
5 X  r       X
6 X   X      X
7 X d X   P  X
8 X   X      X
9 X   X      X
```

```
  XXXXXXXXXXXX

void set(char world[10][10], int x, int y, char c);
Sets world[x][y] to character c.

void clear(char world[10][10], int x, int y);
Sets world[x][y] to ' '.

bool is_unoccupied(char world[10][10], int x, int y);
Returns true exactly when world[x][y] is a space.

bool is_occupied(char world[10][10], int x, int y);
Returns true exactly when world[x][y] is not a space.

**********************************************************************/

#ifndef WORLD_H
#define WORLD_H


#endif
```

Q8. The goal of this question is to implement the `init` and `print` of `world.h`. The implementation is of course placed in `world.cpp`.

The folder for this question includes the files from the previous question.

The algorithm for `init` is at the end of this question. You must follow it.

```
#include <iostream>
#include <cstdlib>
#include "Robot.h"
#include "world.h"

// Test functions from previous questions

void test_init_and_print_world()
{
    char world[10][10];
    init(world);
    print(world);
}


int main()
{
    int seed;
    std::cin >> seed;
    srand(seed);

    int option = 0;
    std::cin >> option;
    switch (option)
    {
        // code from Q1 - Q7.

        case 8:
            test_init_and_print_world();
            break;
    }

    return 0;
}
```

Test 1.

```
8
   0123456789
  XXXXXXXXXXX
0 X          X
1 X  XXXX    X
2 X          X
3 X          X
4 X      X   X
5 X      X   X
6 X      X   X
7 X      X   X
8 X          X
9 X          X
  XXXXXXXXXXX
```

(The actual placement of the walls depend on the seed value of `srand`)

Test 2.

```
9
  0123456789
  XXXXXXXXXXX
0 X          X
1 X          X
2 X  X       X
3 X  X       X
4 X  XXXXX   X
5 X  X       X
6 X          X
7 X          X
8 X          X
9 X          X
  XXXXXXXXXXX
```

Note that the walls do not overlap. For instance the following is *WRONG*:

```
   0123456789
  XXXXXXXXXXX
0 X          X
1 X  XXXX    X
2 X     X    X
3 X     X    X
4 X     X    X
5 X          X
6 X          X
7 X          X
8 X          X
```

```
9 X          X
  XXXXXXXXXXXX
```

In other words, within the world, there *must* be exactly 8 `X`'s.

Algorithm for setting up the two random walls in the world

First randomly pick an integer x from 0 to 9 and do the same to get a random y. This gives you a point in the world which will serve as a starting point for the first wall. Now randomly pick a random integer 0, 1, 2, 3 where 0 means north, 1 means south, 2 means east, and 3 means west. If you picked 0, then starting the above (x, y), you build a wall in the north direction so that it has length 4. If this is impossible (for instance the (x, y) is too close to the left of the world and the direction is west.), you start all over. Similar for the other 3 directions. If this wall is in the world, you continue on to the second wall.

The process for erecting the second wall is the same, except that if it's impossible, you get rid of the first wall altogether and start all over. Note that the second wall has length 4, must stay in the world, and must not overlap the first wall.

Q9. The goal is here to implement and test

```
void set(char world[10][10], int x, int y, char c);
```

Here's the test function to test the `set` function:

```
void test_set()
{
    char world[10][10];
    init(world);
    int x = 0;
    int y = 0;
    char c = ' ';
    std::cin >> x >> y >> c;
    set(world, x, y, c);
    print(world);
}
```

Note that the function simply puts character `c` at the given position `x`, `y` in the 2d array `world`. It does not check if the character at that position is a space or not.

I leave it to you to implement and test the following functions:

```
void clear(char world[10][10], int x, int y);
bool is_unoccupied(char world[10][10], int x, int y);
bool is_occupied(char world[10][10], int x, int y);
```

Skeleton code for power station

Now for the power stations. Here's the header file `PowerStation.h`: a05q09/skel/powerstation.h

```
#ifndef POWERSTATION_H
#define POWERSTATION_H

struct PowerStation
{
    int x, y;
    int energylevel;
};

void init(PowerStation & powerstation, int x, int y, int energylevel);
void print(const PowerStation & powerstation);
void dec_energylevel(PowerStation & powerstation, int d);

#endif
```

Q10. Implement

```
void init(PowerStation & powerstation, int x, int y, int energylevel);
void print(const PowerStation & powerstation);
```

In `main()` add a test function

```
void test_init_and_print_power_station()
{
    int x = 0, y = 0;
    int energylevel;
    std::cin >> x >> y >> energylevel;
    PowerStation ps;
    init(ps, x, y, energylevel);
    print(ps);
}
```

The option that triggers this test is option 10.

Complete the implementation file `PowerStation.cpp`. The `print()` function for `PrintStation.cpp` prints the instance variables in the obvious way. Here's an example

```
<PowerStation: x=5, y=8, energylevel=100>
```

Q11. Now implement and test this:

```
void dec_energylevel(PowerStation & powerstation, int d);
```

Q12. I have given you enough. I did not give you implementation details on the pot of gold. But you have enough to figure it out on your own. (There are many ways to implement the pot of gold. It does not matter how you choose to do it. Just get it done.)

Now complete the problem. As much as possible, you should use functions available.

The general pseudocode in the main program looks like this:

```
build world (including the walls)
create power stations
put power stations at random positions in the world
build robots (three of them with names c, d, r)
put robots at randomly chosen available positions in the world
set time to 100

while time > 0:

    if any robot has found the gold:
        break

    move each robot randomly by one step to an available unoccupied
    position

    for each power station, check which robot(s) are just next to it
    and transfer energy to robot(s) according to the above given rules

    decrement time
```

(There's no fighting among the robots!)

You are strongly encouraged to add your own functions to the various files to make your code readable.

(Note: `r` = r2d2, `c` = c3p0, `d` = Darth Vader. Darth Vader is sort of like a robot ... yes? no?)