Project Three: Simple World

Out: June 18, 2018; Due: July 8, 2018

I. Motivation

1. To give you experience in using arrays, pointers, structs, enums, and different I/O streams and writing program that takes arguments.

2. To let you have fun with an application that is extremely captivating.

II. Introduction

1. Overview

The simple world program we will write for this project simulates a number of creatures running around in a simple world. The world is an *m*-by-*n* two-dimensional grid of **squares** (The number *m* represents the height of the grid and the number *n* represents the width of the grid.). Each creature lives in one of the squares, faces in one of the major compass directions (north, east, south, or west), and belongs to a particular **species**, which determines how that creature behaves. Each square also has a specific terrain type.
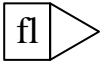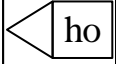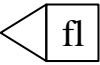

Figure 1. A 4-by-4 grid, which contains five creatures.

Figure 1 shows a 4-by-4 grid populated by five creatures. Two of them belong to the species flytrap (whose short name is "fl"), two belong to the species hop (whose short name is "ho"), and one belongs to the species landmine (whose short name is "la"). The direction of each creature is represented by the direction of the arrow. For example, the flytrap at the top row is facing east and the flytrap at the bottom row is facing west. The character in each square indicates the terrain type of the square. There are four terrain types: plain (P), lake (L), forest (F), and hill (H) (See Section II-3 for more details). Each creature belongs to one species and each species has an associated program which controls how each creature of that species behaves (See Section II-2

for more details). Each creature may also have some special abilities, including flying and archery (See Section II-4 for more details).

Table 1. The list of instructions and their explanations.

hop	The creature moves forward. If moving forward would put the creature outside the boundaries of the grid, would cause the creature **that cannot fly** to land in a "lake" square, **or** would cause the creature to land on top of another creature, the **hop** instruction does nothing.
left	The creature turns left 90 degrees to face in a new direction.
right	The creature turns right 90 degrees to face in a new direction.
infect	If the square immediately in front of this creature is occupied by a creature of a different species (which we refer to as an "**enemy**"), that enemy creature is infected to become the same as the infecting species. When a creature is infected, it keeps its position and orientation, but changes its internal species indicator and begins executing the same program as the infecting creature, starting at step 1. If the square immediately in front of this creature is empty, outside the grid, being a forest square, **or** occupied by a creature of the same species, the **infect** instruction does nothing. If the creature has the archery ability, its infecting action is different (see Section II-4 for more details).
ifempty n	If the square in front of the creature is inside the grid boundary and unoccupied, jump to step n of the program; otherwise, go on with the next instruction in sequence.
ifenemy n	If the square the creature is facing is not a "forest" square and is occupied by a creature of an enemy species, jump to step n; otherwise, go on with the next instruction.
ifsame n	If the square the creature is facing is not a "forest" square and is occupied by a creature of the *same* species, jump to step n; otherwise, go on with the next instruction.
ifwall n	If the creature is facing the border of the grid (which we imagine as consisting of a huge wall), or the creature is facing a lake square and it cannot fly, jump to step n of the program; otherwise, go on with the next instruction

	in sequence.
go *n*	This instruction always jumps to step *n*, independent of any condition.

2. Species Instructions

Each **species** has an associated **program** which controls how each creature of that species behaves. Programs are composed of a sequence of **instructions**. The instructions that can be part of a program are listed in Table 1. There are nine *legal* instructions in total. The last five instructions have an additional integer argument.

Program is an attribute associated with species. Creatures of the same species have the same program. However, different species have different programs.

For example, the program of the species *flytrap* is composed of the following five instructions:

```
ifenemy 4
left
go 1
infect
go 1
```

The meaning of each instruction for this example is commented below:

```
(step 1) ifenemy 4    # If there is an enemy ahead, go to step 4
(step 2) left         # Turn left
(step 3) go 1         # Go to step 1
(step 4) infect       # Infect the adjacent creature
(step 5) go 1         # Go to step 1
```

We will simulate the behaviors of all the creatures for a user specified number of rounds. In each round, creatures take their turns one by one, starting from the first creature. After the first creature finishes its turn, the second creature begins its turn. So on and so forth. One round ends with the last creature finishing its turn. Then the next round begins with the first creature taking its turn. Note that during the simulation, a creature may infect another creature so that the infected one changes its species. However, the simulation order of the infected creature **does not change**.

Each creature also maintains a variable called program counter which stores the index of the instruction it is going to execute. On each turn of a creature, it executes a number of instructions of its program, starting from the step indicated by the program counter. A program ordinarily

continues with each new instruction **in sequence**, although this order can be changed by certain instructions in the program such as the **if***** instructions. In each turn, a creature can execute any number of **if***** or **go** instructions without relinquishing this turn. Its turn ends only when the creature executes one of the instructions: **hop**, **left**, **right**, or **infect**. After its turn ends, the creature updates the program counter to point to the next instruction, which will be executed at the beginning of its next turn.

Note that each creature maintains its own program counter, so that two different creatures belonging to the same species can have different program counters. **The indices of the instructions start from one, i.e., the first instruction of each program is "step 1".** At the very beginning of the simulation process, the program counters of all the creatures are set to their first instructions.

3. Terrain Information

In order to simulate the geographical environment of the world, the simple world has the following four terrain types. Each square in the grid belongs to one of these four types.

1. **Plain (P)**: A square of the plain type is represented as **P** in the grid. A plain square does not have any side effects to a creature in it.

2. **Lake (L)**: A square of the lake type is represented as **L** in the grid. If a creature cannot fly, it cannot move into a lake square; otherwise, it can move into a lake square. Therefore, a lake square is treated as an impassible wall for a creature that cannot fly. In summary, if a creature is facing a lake and it cannot fly, the ifwall instruction should return true. If the creature is facing a lake and it can fly, the ifwall instruction should return false.

3. **Forest (F)**: A square of the forest type is represented as **F** in the grid. When a creature is in a forest square, it cannot be seen by another creature. Therefore, a creature in a forest square cannot be infected by an enemy creature, unless the enemy has the archery ability (see Section II-4 for more details on the archery ability). If a creature (no matter whether it has the archery ability or not) is facing another creature that is in a forest square, the ifempty, ifsame, and ifenemy instructions should all return false.

4. **Hill (H)**: A square of the hill type is represented as **H** in the grid. In a hill square, it is rocky and by default, a creature in that square cannot act quickly. Therefore, a creature in a hill square can do simulation once every two rounds. For example, if a creature just moves from a non-hill square into a hill square in one round (assuming that this is the first round), it should stay idle (i.e., do not execute any instructions) in the second round. Then, in the third round, the creature continues to simulate the next instruction. If, by the end of the third round, the creature still stays in a hill square, it stays idle in the fourth round. It simulates the next instruction in the fifth round. On the other hand, if, by the end of the third round, the

creature is in a non-hill square, it simulates the next instruction in the fourth round. Note that if the creature is in the hill square initially, it should be treated as staying idle in the first round; it simulates the next instruction in the second round. Also note that the side effect of a hill square is not applied to a creature that can fly.

4. Special Ability of Creature

Each creature might have **special abilities** (hereafter, referred to as **abilities**) in the simple world. There are two kinds of abilities: flying (f) and archery (a). **Creatures may have no abilities. They may also have multiple abilities. The abilities of a creature will be maintained throughout the whole simulation, even if the creature is infected.** Creatures of the same species may have different sets of abilities. Below are the details of these two abilities.

1. **Flying (f):** A creature with the flying ability can override the restriction of a lake square and a hill square. It can move into a lake square. It can simulate its behavior once every round even if it is in a hill square. If it is in a forest square, the effect of a forest square on it does not change.

2. **Archery (a):** For a creature *A* with the archery ability, if it infects, it will infect the first creature of an **enemy** species (if existing) in its facing direction no matter how far they are away from each other and no matter whether there are creatures of the same species as *A* in between them. A creature with archery ability can see enemies in forest. Therefore, **it can infect the enemy even if the enemy is in a forest square.**

III. Available Types

In completing this project, you will have the following types available to you. They are defined in the file `world_type.h`.

```
const unsigned int MAXSPECIES = 10; // Max number of species in the
                                    // world
const unsigned int MAXPROGRAM = 40; // Max size of a species program
const unsigned int MAXCREATURES = 50; // Max number of creatures in
                                      // the world
const unsigned int MAXHEIGHT = 20; // Max height of the grid
const unsigned int MAXWIDTH = 20;  // Max width of the grid


enum direction_t { EAST, SOUTH, WEST, NORTH, DIRECT_SIZE };
```

```
/*
// Type: direction_t
// ----------------
// This type is used to represent direction, which can take one of
// the four values: East, South, West, and North. The last one,
// DIRECT_SIZE, can be used to indicate the end of this enum type.
// This convention is applied to the other enum types defined below.
*/

enum terrain_t { PLAIN, LAKE, FOREST, HILL, TERRAIN_SIZE };
/*
// Type: terrain_t
// ----------------
// This type is used to represent terrain type of a square, which
// can take one of the four values: Plain, Lake, Forest, Hill.
*/

enum ability_t { FLY, ARCH, ABILITY_SIZE };
/*
// Type: ability_t
// ----------------
// This type is used to represent special abilities of a creature,
// which can take one of the two values: Flying and Archery.
*/

enum opcode_t { HOP, LEFT, RIGHT, INFECT, IFEMPTY, IFENEMY, IFSAME,
IFWALL, GO, OP_SIZE };
/*
// Type: opcode_t
// -------------
// The type opcode_t is an enumeration of all of the legal
// command names.
*/

struct point_t
{
    int r;
    int c;
};
/*
// Type: point_t
// ------------
// This type is used to represent a point in the grid.
// Its component r corresponds to the row number; its component
// c corresponds to the column number;
```

```
*/

const string directName[] = {"east", "south", "west", "north"};
// An array of strings representing the direction name.

const string directShortName[] = {"e", "s", "w", "n"};
// An array of strings representing the short name of direction.

const string terrainName[] = {"plain", "lake", "forest", "hill"};
// An array of strings representing the terrain type name.

const string terrainShortName[] = {"P", "L", "F", "H"};
// An array of strings representing the short name of terrain type.

const string abilityName[] = {"fly", "arch"};
// An array of strings representing the ability name.

const string abilityShortName[] = {"f", "a"};
// An array of strings representing the short name of ability.

const string opName[] = {"hop", "left", "right", "infect",
    "ifempty", "ifenemy", "ifsame", "ifwall", "go"};
// An array of strings representing the command name.

struct instruction_t
{
    opcode_t op;
    unsigned int address;
};
/*
// Type: instruction_t
// ------------------
// The type instruction_t is used to represent an
// instruction and consists of a pair of an operation
// code and an integer. For some operation code, the
// integer stores the address of the instruction it jumps
// to (e.g., n in the instruction "ifempty n"). The
// address is optional.
*/

struct species_t
{
    string name;
    unsigned int programSize;
    instruction_t program[MAXPROGRAM];
```

```
};
/*
// Type: species_t
// ------------------
// The type species_t is used to represent a species
// and consists of a string, an unsigned int, and an array
// of instruction_t. The string gives the name of the
// species. The unsigned int gives the number of instructions
// in the program of the species. The array stores all the
// instructions in the program according to their sequence.
*/

struct creature_t
{
    point_t location;
    direction_t direction;
    species_t *species;
    unsigned int programID;
    bool ability[ABILITY_SIZE];
    bool hillActive;
};
/*
// Type: creature_t
// ------------------
// The type creature_t is used to represent a creature. It
// consists of a point_t, a direction_t, a pointer to
// species_t, an unsigned int, a bool array, and a bool variable.
// The point_t variable location gives the location of the
// species. The direction_t variable direction gives the direction
// of the species. The pointer to species_t variable species
// points to the species the creature belongs to. The unsigned
// int programID gives the index of the instruction to be executed
// in the instruction_t array of the species. The bool array
// ability indicates the set of abilities the creature has. For
// example, if the creature only has FLY ability, you should set
// ability[FLY] to true and ability[ARCH] to false. The bool
// variable hillActive indicates if the creature is able to simulate
// in the current round if the creature is in a hill square. It does
// not serve any other purpose if the creature is not in a hill
// square.
*/

struct grid_t
{
    unsigned int height;
```

```
    unsigned int width;
    creature_t *squares[MAXHEIGHT][MAXWIDTH];
    terrain_t terrain[MAXHEIGHT][MAXWIDTH];
};
/*
// Type: grid_t
// ------------------
// The type grid_t consists of the height and the width of
// the grid, a two-dimensional array of pointers to creature_t,
// and a two-dimensional array of terrain_t type. If there is
// a creature at the point (r, c) in the grid, then squares[r][c]
// stores a pointer to that creature. If point (r, c) is not
// occupied by any creature, then squares[r][c] is a NULL pointer.
// The two-dimensional array terrain stores the terrain type for
// each square in the grid.
*/

struct world_t
{
    unsigned int numSpecies;
    species_t species[MAXSPECIES];

    unsigned int numCreatures;
    creature_t creatures[MAXCREATURES];

    grid_t grid;
};
/*
// Type: world_t
// --------------
// This type consists of two unsigned ints, an array of species_t,
// an array of creature_t, and a grid_t object. The first unsigned
// int numSpecies specifies the number of species in the creature
// world. The second unsigned int numCreatures specifies the number
// of creatures in the world. All the species are stored in the array
// species and all the creatures are stored in the array creatures.
// The grid is given in the object grid.
*/
```

IV. File Input

All the species, the programs for all the species, and the initial layout of the creature world are stored in files and these files will be read by your program to set up the simulation environment. **Note**: when you read files, **you must use input file stream `ifstream`**. Otherwise, since the files are read-only on our online judge, you may fail to read the files.

As we described before, each species has an associated program. The program for each species is stored in a separate file whose name is just the name of that species. For example, the program for the species *flytrap* is stored in a file called `flytrap`.

A file describing a program contains all the instructions of that program in order. Each line lists just one instruction. The first line lists the first instruction; the second line lists the second instruction; so on and so forth. Each instruction is one of the nine legal instructions described in Table 1. The program ends with the end of file or a blank line. Comments may appear after the blank line or at the end of each instruction line. For example, the program file for the flytrap species looks like:

```
ifenemy 4    If there is an enemy, go to step 4.
left         If no enemy, turn left.
go 1
infect
go 1

The flytrap sits in one place and spins.
It infects anything which comes in front.
Flytraps do well when they clump.
```

Note that in writing functions for reading these program files, you should handle the comments correctly, which means that you should ignore these comments when setting up the program for a species.

Since there are many species, we stored all of their program files in a directory.

To help you get all the species and their program files, we also have a file telling the directory where the program files are stored and listing all the species. We call this file a species summary. The first line of this file shows the directory where all of the program files are stored. The next lines list all the species, with one species per line. For example, the following is a species summary file:

```
creatures
flytrap
```

```
hop
landmine
```

From this file, we can learn that the program files are stored in the directory called `creatures` within the current working directory. We have three species to simulate, which are *flytrap*, *hop*, and *landmine*. By first reading the species summary file, you will know where to find the program file for each species.

Finally, there is a file describing the initial state of the creature world. We call it a world file. The first line of this file gives the height of the two-dimensional grid (i.e., the number of rows) and the second line gives the width of the grid (i.e., the number of columns). Afterwards, it is the terrain layout of the grid. The terrain layout is indicated by a two-dimensional array of characters, with each character being one of 'P', 'L', 'F', and 'H', corresponding to plain, lake, forest, and hill types, respectively. The remaining lines of this file describe all the creatures to simulate and their initial directions, locations, and abilities, with one creature per line. Each of these lines has the following format:

```
<species> <initial-direction> <initial-row> <initial-column>
[ability-1] [ability-2]
```

where `<species>` is one of the species from the species summary file, `<initial-direction>` describes the initial direction and is one of the strings "**east**", "**south**", "**west**", and "**north**". `<initial-row>` describes the initial row location of the creature. We use the convention that **the top-most row of the grid is row 0 and the row number increases from top to bottom.** `<initial-column>` describes the initial column location of the creature. We use the convention that **the left-most column of the grid is column 0 and the column number increases from left to right.** `[ability-i]` is optional. It describes the *i*-th ability of the creature and is one of the characters 'f' and 'a', corresponding to the flying and archery abilities, respectively. Note there could be no ability entry. There is no specific order on which type should appear first.

An example of a world file looks like:

```
4
4
PPLL
PPLL
FFHH
FFHH
hop east 2 0 f a
flytrap east 2 2
```

It says that the size of the grid is 4-by-4 and the top left, top right, bottom left, and bottom right quadrants are plains, lakes, forests, and hills, respectively. There are two creatures in the world. The first creature belongs to the species *hop*. It faces east and lives at point (2, 0) initially. It has the abilities of flying and archery. The second creature belongs to the species *flytrap*. It faces east and lives at point (2, 2) initially. It has no special abilities.

In the simulation, the order on the creatures to simulate is important. This order is determined by the order that these creatures appear in the world file.

V. Program Arguments

Your program will obtain the names of the species summary file and the world file via program arguments. Additionally, your program will be told the number of rounds to simulate and whether it should print the simulation result verbosely or not.

The expected order of arguments is:

```
<species-summary> <world-file> <rounds> [v|verbose]
```

The first three arguments are mandatory. They give the name of the species summary file, the name of the world file, and the number of simulation rounds, respectively. The fourth argument is optional. If the fourth argument is the string "v" or the string "verbose", your program should print the simulation result verbosely, which will be explained later. Otherwise, if it is omitted or is any other string, your program should print the result concisely, which will also be explained later. If there are more than four arguments, the remaining arguments are ignored.

Suppose that you program is called p3. It may be invoked by typing in a terminal:

```
./p3 species world 10 v
```

Then your program should read the species summary from the file called "species" and the world file from the file called "world". The number of simulation rounds is 10. Your program should print the simulation information verbosely.

VI. Error Checking and Error Message

Your program should check for errors **before** it starts to simulate the moves of the creatures. If any error happens, your program should issue an error message and then exit. If there are no errors happening, then the initial state of the creature world is legal and your program can start simulating the simple world.

We require you to do the following error checking and print the error message in **exactly** the same way as described below. Note that some of the output error message has two lines and each error message should be ended with a newline character. All error messages should be sent to the standard output stream `cout`; none to the standard error stream `cerr`.

1. Check whether the number of arguments is less than three. If it is less than three, then one of the mandatory arguments is missing. You should print the following error message:

```
Error: Missing arguments!
Usage: ./p3 <species-summary> <world-file> <rounds> [v|verbose]
```

2. Check whether the value `<rounds>` supplied by the user is negative. If it is negative, you should print the following error message:

```
Error: Number of simulation rounds is negative!
```

3. Check whether file open is successful. If opening species summary file, world file, *or* any species program file fails (for example, the file to be opened does not exist), print the following error message:

```
Error: Cannot open file <filename>!
```

where `<filename>` should be replaced by the name of the file that fails to be opened. If that file is not in the same directory as your program, you need to include its path in the `<filename>`. As you may know, there are multiple ways to specify a path. For us, the path name should be specified in the most basic way, i.e., "`<dir>/<filename>`" (not "`./<dir>/<filename>`", "`<dir>//filename`", etc.). Once you find a file that cannot be opened, issue the above error message and terminate your program.

4. Check whether the number of species listed in the species summary file exceeds the maximal number of species `MAXSPECIES`. If so, print the following error message:

```
Error: Too many species!
Maximal number of species is <MAXSPECIES>.
```

where `<MAXSPECIES>` should be replaced by the maximal number of species set by your program.

5. Check whether the number of instructions for a species exceeds the maximal size of a species program `MAXPROGRAM`. If so, print the following error message:

```
Error: Too many instructions for species <SPECIES_NAME>!
Maximal number of instructions is <MAXPROGRAM>.
```

where `<SPECIES_NAME>` should be replaced by the name of the species whose program has more instructions than the maximal number allowed and `<MAXPROGRAM>` should be replaced by the maximal size of a species program set by your program.

6. Check whether the instructions are valid. The species program file contains instructions. We only allow nine instructions as listed in Table 1. Your program needs to check whether the instruction name is one of the nine legal instruction names listed in the string array `opName` (which is defined in Section III). If an instruction name is not recognized, you should print the following error message:

```
Error: Instruction <UNKNOWN_INSTRUCTION> is not recognized!
```

where `<UNKNOWN_INSTRUCTION>` should be replaced by the name of the unrecognized instruction. You can assume that for any recognized instruction, it is given in the correct format. Thus, you don't need to check whether an integer is appended after the instruction name or not. If there are multiple unrecognized instruction names, you only need to print out the first one and then terminate the program.

7. Check whether the grid height given by the world file is legal. A legal grid height is at least **ONE** and less than or equal to a maximal value `MAXHEIGHT`. If the grid height is illegal, print the following error message:

```
Error: The grid height is illegal!
```

8. Check whether the grid width given by the world file is legal. A legal grid width is at least **ONE** and less than or equal to a maximal value `MAXWIDTH`. If the grid width is illegal, print the following error message:

```
Error: The grid width is illegal!
```

9. Check whether the two-dimensional array describing the terrain layout is correct. You can assume that the height and the width of the array are equal to the correct values. You only need to check whether each character is the array is one of the valid characters, 'P', 'L', 'F', and 'H'. If a character is not valid, print the following error message:

```
Error: Terrain square (<CHAR> <R> <C>) is invalid!
```

where `<R>` should be replaced by the row location of the invalid square, `<C>` be replaced by the column location of the square, , and `<CHAR>` be replaced by the invalid character in the square. For example, given the following world file:

```
2
2
PP
PA
flytrap east 0 0 f a
food west 1 1 f
```

You should print the error message:

```
Error: Terrain square (A 1 1) is invalid!
```

If there are multiple invalid characters, you only need to print out the first one and then terminate the program. The first invalid character has the smallest row number, and among all the invalid characters with that same smallest row number, the first invalid character has the smallest column number.

10. Check whether the number of creatures listed in the world file exceeds the maximal number of creatures `MAXCREATURES`. If so, print the following error message:

```
Error: Too many creatures!
Maximal number of creatures is <MAXCREATURES>.
```

where `<MAXCREATURES>` should be replaced by the maximal number of creatures allowed by your program.

11. Check whether each creature in the world file belongs to one of the species listed in the species summary file. If the species for a creature is not recognized, print the following error message:

```
Error: Species <UNKNOWN_SPECIES> not found!
```

where `<UNKNOWN_SPECIES>` should be replaced by the unrecognized species. If there are multiple unrecognized species, you only need to print out the first one and then terminate the program.

12. Check whether the direction string for each creature in the world file is one of the strings in the array `directName` (which is defined in Section III). If the direction string is not recognized, print the following error message:

```
Error: Direction <UNKNOWN_DIRECTION> is not recognized!
```

where `<UNKNOWN_DIRECTION>` should be replaced by the unrecognized direction name. If there are multiple unrecognized direction names, you only need to print out the first one and then terminate the program.

13. Check whether each creature is inside the boundary of the grid. If any creature is outside the boundary, print the following error message:

```
Error: Creature (<SPECIES> <DIR> <R> <C>) is out of bound!
The grid size is <HEIGHT>-by-<WIDTH>.
```

where `<SPECIES>` should be replaced by the species the creature belongs to, `<DIR>` be replaced by the direction the creature is facing, `<R>` be replaced by the row location of the creature, `<C>` be replaced by the column location of the creature, `<HEIGHT>` be replaced by the height of the grid, and `<WIDTH>` be replaced by the width of the grid. Here, we use the four-tuple (`<SPECIES> <DIR> <R> <C>`) to identify the creature. For example, if given the following world file:

```
3
3
PPP
PPP
PPP
flytrap east 0 0 f a
hop south 3 2 a
food west 2 1
```

then, the creature (`hop south 3 2`) is outside the boundary. Then, the error message should be:

```
Error: Creature (hop south 3 2) is out of bound!
The grid size is 3-by-3.
```

If there are multiple creatures outside the boundary, you only need to print out the first one and then terminate the program.

14. Check whether each ability of each creature is valid. If an ability is neither of the two valid characters, 'f' and 'a', print the following error message:

```
Error: Creature (<SP> <DIR> <R> <C>) has an invalid ability <ABILITY>!
```

where the four-tuple (`<SP> <DIR> <R> <C>`) identifies the creature that has an invalid ability, and `<ABILITY>` should be replaced by the invalid ability string. Note that anything that is not one of the character 'f' and 'a' is invalid. This includes strings that do not contain whitespaces but have more than 1 character. For example, for the following world map:

```
2
2
PP
PP
hop south 1 1 kt
```

You should print error message:

```
Error: Creature (hop south 1 1) has an invalid ability kt!
```

If there are multiple such errors, you only need to print out the first invalid ability of the first creature that has invalid abilities. Note that it is allowed to have multiple characters of the same valid ability. You just ignore the redundant ones. For example, creature description

```
hop south 1 1 f a f
```

has two 'f's, but it is legal.

15. Check whether each square in the grid is occupied by at most one creature. If any square is occupied by more than one creature, print the following error message:

```
Error: Creature (<SP1> <DIR1> <R> <C>) overlaps with creature (<SP2> <DIR2> <R> <C>)!
```

where (`<R>` `<C>`) identifies the square which is occupied by more than one creature, the first four-tuple (`<SP1>` `<DIR1>` `<R>` `<C>`) identifies **the second creature in order** that occupies the square, and the second four-tuple (`<SP2>` `<DIR2>` `<R>` `<C>`) identifies **the first creature in order** that occupies the square. Once you find two creatures occupying the same square, you issue the above error message and then terminate the program.

16. Check whether a creature that cannot fly is not in a lake square. If a creature that cannot fly is in a lake square, print the following error message:

```
Error: Creature (<SP> <DIR> <R> <C>) is in a lake square!
```
```
The creature cannot fly!
```

where the four-tuple (`<SP>` `<DIR>` `<R>` `<C>`) identifies the creature that cannot fly but is in a lake square. If there are multiple such creatures, you only need to print out the first one and then terminate the program.

Since you may implement the error checking in different order and in the case that there is more than one error, the first error message printed out may be different. Therefore, we will only test your error checking using test cases containing just one error.

VII. Simulation Output

Once all of the above error checkings on the initial state of the creature world are passed, you can start simulating the creature world. You should print to the standard output the simulation information, either in a verbose mode or in a concise mode, **depending on whether an additional argument "v" or "verbose" is provided by the user.**

In the verbose mode, you first print the initial state of the world. In doing so, you begin with printing the string "`Initial state`" followed by a newline. Then you graphically show the layout of the initial grid using just characters. Each square takes a four-character field in your terminal. Adjacent squares on the same row are separated by one space. If a square in the grid is not occupied by any creature, the field for that square is filled with **FOUR** "_". If a square is occupied by a creature, then the first two characters of the field for that square are the first two letters of the name of the species the creature belongs to. (We assume that all the species names contain at least two characters and no two species names have the identical first two characters.) The third character in the field is a "_" and the fourth character is the first character of the direction the creature faces, i.e., "e" for "east", "s" for "south", "w" for "west", and "n" for "north".

For example, suppose a world file looks like

```
4
4
PPPP
PPPP
PPPP
PPPP
hop east 2 0
flytrap east 2 2
```

(Note that the above example is a trivial example, in which all the terrain squares are plain and no creatures have special abilities. However, it is enough to illustrate the output format.)

Then the layout of the initial grid is printed as

```
____ ____ ____ ____
____ ____ ____ ____
ho_e ____ fl_e ____
____ ____ ____ ____
```

Note that there is a space at the end of each line.

After printing the initial layout, we begin the simulation from the first round to the last round specified by the user. In the *i*-th simulation round, you first print "Round <i>" followed by a newline. For example, in the first round, you should first print

```
Round 1
```

During each simulation round, you simulate the moves of all the creatures in turn. When starting simulating a creature, you announce that this creature takes action by printing

```
Creature (<SPECIES> <DIR> <R> <C>) takes action:
```

followed by a newline. In the above output, the four-tuple (<SPECIES> <DIR> <R> <C>) shows the state of the creature right before it takes the action, where <SPECIES> should be replaced by the species the creature belongs to, <DIR> be replaced by the direction the creature is facing, <R> be replaced by the row location of the creature, and <C> be replaced by the column location of the creature.

After this, you print the sequence of instructions that the creature executes for its turn. This sequence may include any number of **if***** and **go** instructions and end with one of the **hop**, **left**, **right**, and **infect** instruction. You should print the sequence of instructions the creature executes in order, with one instruction per line. The output format for an instruction is:

```
Instruction <INSTR_NO>: <INSTR_NAME> [GOTO_STEP]
```

where <INSTR_NO> should be replaced by the number of that instruction in the program (the number starts from 1), <INSTR_NAME> should be replaced by the name of the instruction, and [GOTO_STEP] is the number in an **if***** or a **go** instruction and is optional.

After printing the last instruction of the creature under consideration, you should print the updated layout of the grid, using the same rule as you print the initial layout.

In the special case where the creature is in a hill square and stays idle for the current round, then nothing should be printed out for this creature, including the layout of the grid.

Now let's look at an example. Suppose that the program for the species `hop` is

```
hop
go 1
```

and the program for the species `flytrap` is

```
ifenemy 4    If there is an enemy, go to step 4.
left         If no enemy, turn left.
go 1
infect
go 1
```

Then, given the following world file

```
4
4
PPPP
PPPP
PPPP
PPPP
hop east 2 0
flytrap east 2 2
```

the simulation information for the first round is printed as

```
Round 1
Creature (hop east 2 0) takes action:
Instruction 1: hop

____ ____ ____ ____

____ ____ ____ ____

____ ho_e fl_e ____

____ ____ ____ ____

Creature (flytrap east 2 2) takes action:
Instruction 1: ifenemy 4
Instruction 2: left

____ ____ ____ ____

____ ____ ____ ____

____ ho_e fl_n ____

____ ____ ____ ____
```

The simulation information for the second round is printed as

```
Round 2
Creature (hop east 2 1) takes action:
Instruction 2: go 1
Instruction 1: hop

____ ____ ____ ____

____ ____ ____ ____

____ ho_e fl_n ____

____ ____ ____ ____

Creature (flytrap north 2 2) takes action:
Instruction 3: go 1
Instruction 1: ifenemy 4
```

```
Instruction 2: left
```

____ ____ ____ ____

____ ____ ____ ____

____ ho_e fl_w ____

____ ____ ____ ____

In the concise mode, you print the initial state of the world in the same way as in the verbose mode. When printing the simulation information for the *i*-th round, you first print "Round <i>" followed by the newline. Then you print the **_final_** action of each creature in turn, with one creature per line. The format is:

```
Creature (<SPECIES> <DIR> <R> <C>) takes action: <LAST_INSTR>
```

Same as in the verbose mode, the four-tuple (<SPECIES> <DIR> <R> <C>) shows the state of the creature right before it takes the action. <LAST_INSTR> should be replaced by the last instruction the creature executes for its turn, which is one of the **hop**, **left**, **right**, and **infect** instruction. In the special case where the creature is in a hill square and stays idle for the current round, then nothing should be printed out for this creature.

After printing the final actions for all the creatures, you print the updated layout **at the end of this round**.

For the same world file as above:

```
4
4
PPPP
PPPP
PPPP
PPPP
hop east 2 0
flytrap east 2 2
```

In the concise mode, the simulation information for the first round is printed as

```
Round 1
Creature (hop east 2 0) takes action: hop
Creature (flytrap east 2 2) takes action: left
```

____ ____ ____ ____

____ ____ ____ ____

____ ho_e fl_n ____

____ ____ ____ ____

The simulation information for the second round is printed as

```
Round 2
Creature (hop east 2 1) takes action: hop
Creature (flytrap north 2 2) takes action: left

____ ____ ____ ____
____ ____ ____ ____
____ ho_e fl_w ____
____ ____ ____ ____
```

There are no blank lines in the output for both the verbose and concise mode.

VIII. Source Code Files and Compiling

There is a source code file located in the Project-3-Related-Files.zip on Canvas:

`world_type.h`: The header file which defines a number of types for you to use.

You should copy this file into your working directory. **DO NOT modify it!**

You need to write three other source code files. The first file is named as `simulation.h`, which contains the declarations for all the functions you write, just like the `p2.h` in our project two. The second file is named as `simulation.cpp`, which contains all the implementations of those functions declared in the `simulation.h`. The third file is named as `p3.cpp`, which contains only the main function. You should also write a `Makefile` for compiling the final program. **The final program must be named p3.**

IX. Implementation Requirements and Restrictions

1. In writing your code, you may use the following standard header files: `<iostream>`, `<fstream>`, `<sstream>`, `<iomanip>`, `<string>`, `<cstdlib>`, and `<cassert>`. No other header files can be included.

2. You may **not** define any global variables yourself. You can only use the global constant ints and string arrays defined in `world_type.h`.

3. Pass large structs by reference rather than value. Where appropriate, pass const references / pointers-to-const. Do not pass lots of little arguments when you can pass an appropriate, larger structure instead.

4. All required output should be sent to the standard output stream; none to the standard error stream.

5. You should strive not to duplicate identical or nearly-identical code, and instead collect such code into a single function that can be called from various places. Each function should do a single job, though the definition of "job" is obviously open to interpretation. Most students write too few functions that are too large.

X. Hints and Tips

1. This project will take you longer than project three did, so **start early**!

2. Do this project in stages. First, be able to read the species summary file. Second, be able to read the programs for all the species. Third, be able to read the world file. Write some diagnostic code that can print out the species summary, the program for each species, and the creatures, to

make sure that you are reading them correctly. Implement the error checking and test it with different illegal inputs. Once you can read the structures in, implement the simple moves such as **left** and **right**. Once you have that working, implement moves such as **hop** and **infect**. Finally, implement **if***** and **go** instructions.

3. Take advantage of the fact that enumerations are sequentially numbered from 0 to N-1.

4. Use the right methods of input file stream to read file. In some cases, you may first use the `getline()` function to read the entire line of a file and then use an input string stream to extract the content from that line.

5. The **hop** instruction will only cause the creature to move forward when the square it is facing is empty. If moving forward would put the creature outside the boundaries of the grid, would cause the creature that cannot fly to land in a "lake" square, or would cause the creature to land on top of another creature, the **hop** instruction does nothing. However, although the hop action is not executed successfully, you should update the program counter so that it points to **<u>the next</u>** instruction after this hop instruction. The similar situation also applies to the **infect** instruction. If there is no enemy to infect, the infect operation does nothing. However, you should update the program counter to its next instruction.

6. As a hint, you probably need to write the following eight functions or some variations of them. However, these are not the only functions you have to write. You probably need to write more functions for different jobs.

```
bool initWorld(world_t &world, const string &speciesFile,
    const string &worldFile);
// MODIFIES: world
//
// EFFECTS: Initialize "world" given the species summary file
//          "speciesFile" and the world description file
//          "worldFile". This initializes all the components of
//          "world". Returns true if initialization is successful.


void simulateCreature(creature_t &creature, grid_t &grid, bool
verbose);
// REQUIRES: creature is inside the grid.
//
// MODIFIES: creature, grid, cout.
//
// EFFECTS: Simulate one turn of "creature" and update the creature,
//          the infected creature, and the grid if necessary.
//          The creature programID is always updated. The function
//          also prints to the stdout the procedure. If verbose is
```

```
//              true, it prints more information.

void printGrid(const grid_t &grid);
// MODIFIES: cout.
//
// EFFECTS: print a grid representation of the creature world.

point_t adjacentPoint(point_t pt, direction_t dir);
// EFFECTS: Returns a point that results from moving one square
//          in the direction "dir" from the point "pt".

direction_t leftFrom(direction_t dir);
// EFFECTS: Returns the direction that results from turning
//          left from the given direction "dir".

direction_t rightFrom(direction_t dir);
// EFFECTS: Returns the direction that results from turning
//          right from the given direction "dir".

instruction_t getInstruction(const creature_t &creature);
// EFFECTS: Returns the current instruction of "creature".

creature_t *getCreature(const grid_t &grid, point_t location);
// REQUIRES: location is inside the grid.
//
// EFFECTS: Returns a pointer to the creature at "location" in "grid".
```

XI. Testing

We provide you with a few test cases in the directory called `tests`, which can be found inside Project-3-Related-Files.zip on Canvas.

Inside the `tests` directory, there is an example species summary file called `species` and two subdirectories called `creatures` and `world-tests`. The `creatures` directory contains a number of species program files. The `world-tests` directory contains five world files and the files recording the correct outputs.

The first world file is called `outside-world`, which describes an illegal world with one creature located outside the boundary of the grid.

To run this test case, type the following commands (Note: the name of the compiled program is `p3`):

```
./p3 species world-tests/outside-world 5 > outside-world.out
```

Then the output of your program is redirected to a file named `outside-world.out`. The correct output is recorded in the file `outside-world.answer` in the directory `world-tests`. You can use the `diff` command to check whether the file `outside-world.out` is same as the file `outside-world.answer`.

The second world file is called `overlap-world`, which describes an illegal world with two creatures located at the same square in the grid. You can run this test case using a similar command as shown above and compare your output with the correct output recorded in the file `overlap-world.answer`.

The next three world files `world1`, `world2`, and `world3` are legal world files. You can run these test cases in the similar way. The number of simulation rounds for `world1`, `world2`, and `world3` are 5, 20, and 40, respectively. For these test cases, we provide you with both the verbose and the concise output files. The verbose output files are these files named as `*-verbose.answer` and the concise output files are these files named as `*-concise.answer`.

These are the minimal amount of tests you should run to check your program. Those programs that do not pass these tests are not likely to receive much credit. You should also write other different test cases yourself to test your program extensively. In doing so, you need to write your own legal/illegal species summary files, legal/illegal world files, and species program files. Indeed, it will be very interesting to create new species yourself and observe what kind of species will finally dominate the **SIMPLE WORLD** given different initial layout!

XII. Submitting and Due Date

You should submit your source code files `simulation.h`, `simulation.cpp`, and `p3.cpp`, and the `Makefile`. These files should be submitted as a tar file via the online judgment system. See announcement from the TAs for details about submission. The due date is 11:59 pm on July 8th, 2018.

XIII. Grading

Your program will be graded along three criteria:

1. Functional Correctness

2. Implementation Constraints

3. General Style

Functional Correctness is determined by running a variety of test cases against your program, checking against our reference solution. We will grade Implementation Constraints to see if you have met all of the implementation requirements and restrictions. General Style refers to the ease with which TAs can read and understand your program, and the cleanliness and elegance of your code. For example, significant code duplication will lead to General Style deductions.