COMP 2012 Object-Oriented Programming and Data Structures

Assignment 2 Ecosystem Simulator

Introduction

In this assignment, we will build a program to simulate the evolution of a small ecosystem. The actual ecosystem in nature can be extremely complicated, with hundreds of different species co-existing, cooperating or destroying each other. We will implement a much smaller version of such an ecosystem with only a few animals, but still with elements of real-life nature, such as prey-predator interaction. You will practice using inheritance and polymorphism in this exercise to implement different classes representing various species, each with its own distinct behaviour and overlapping properties described in the base classes.

For this assignment, we will implement 3 species: **Grass**, **Sheep** and **Wolf**. Grasses spread to adjacent tiles every few steps, and are destroyed when animals step on their tiles. Sheep eat grass in order to survive and breed, and move around randomly. Wolves chase and eat sheeps, also in order to survive and breed.

To get a feel for this assignment, you can play around with this simulation and the 3 described species. This simulation is quite different from what we will implement, mainly in terms of how species progress into the next time step; but otherwise, you can mess around and see how the system evolves with different starting conditions (for example, if there were too many wolves in the beginning, or the grass spreads too slowly, etc.)

Class structure overview

Menu

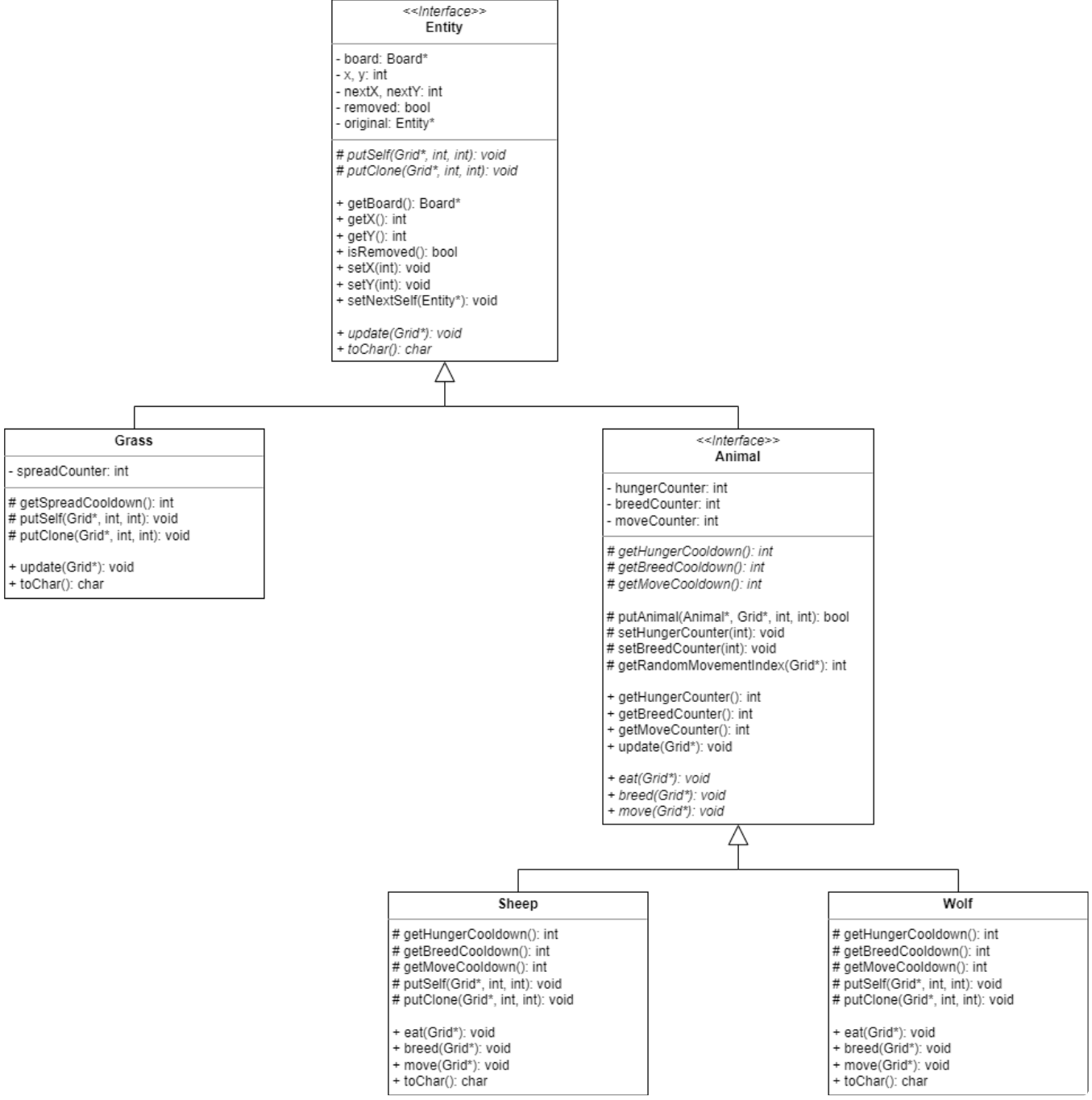
Page maintained by

DINH, Anh Dung
Email: dzung@ust.hk

Last Modified:
11/12/2022 15:28:39

Homepage

Course Homepage



UML Class Diagram for PA2

Because the 3 species share certain similar functions (each step needs to be updated, etc.), we make use of inheritance and polymorphism to define base classes for the species. More specifically, we have designed **Abstract Base Classes (ABC)** - base classes holding shared properties and functions of derived classes, but no objects of such classes can be initialized. These classes have **pure virtual** functions defined, meaning that the derived classes are expected to implement/override them.

The ABC **Entity** is the "root" class: all species will inherit this class. We maintain a grid of Entities, and at each time step, we need to `update()` them and put the corresponding Entities into the grid for the next time step. Functions `putSelf()` and `putClone()` will be implemented by the derived classes to help with placing new Entities on the new grid. And each Entity needs to be printed with a character, but only the derived classes can implement this function with the corresponding character.

Grass inherits **Entity** directly and adds a counter to track when it should "spread" to the adjacent tiles. Meanwhile, **Animal** is another ABC, describing the shared behaviour of sheep and wolves: all animals need to eat, breed if possible, and finally move according to their scheme. Finally, **Sheep** and **Wolf** implement the remaining pure virtual functions after inheriting **Animal**.

Two additional classes are used to maintain the board state: **Board** keeps track of the time steps, and call `update()` on every Entities at each time step, and helps print the board. **Grid** is a convenient class to hold a 2D array of Entities, with boundary check during element access and modification.

Additional remarks

Read the FAQ page for some common clarifications. You should check that a day before the deadline to ensure you don't miss any clarification, even if you have already submitted your work.

If you need further clarification of the requirements, please feel free to post on the Piazza (via Canvas) with the pa2 tag. However, to avoid cluttering the forum with repeated/trivial questions, please **read all the given code, webpage description, sample output, and latest FAQ (refresh this page regularly) carefully before posting your questions**. Also, please be reminded that we won't debug any student's assignment for fairness.

Submission details are in the Submission and Deadline section.

We value academic integrity very highly. Please read the Honor Code section on our course webpage to ensure you understand what is considered plagiarism and the penalties. The following are some of the highlights:

- Do NOT try your "luck" - we use sophisticated plagiarism detection software to find cheaters. We also review codes for potential cases manually.
- The penalty (for **BOTH** the copier and the copiee) is not just getting a zero in your assignment. Please read the Honor Code thoroughly.
- Serious offenders will fail the course immediately, and there may be additional disciplinary actions from the department and university, including expulsion.

Description

Program overview

The program expects input describing the initial board state and the number of turns to simulate. Example input can be seen in `input.txt`:

```
======================
|........S...       W|
|......S.SS...        |
|..............       |
|...........    .     |
|....S......   ...    |
|....S..... .   .     |
|  S.. .. .           |
|  .....SS..          |
|.    ..SS....        |
|      ......       W|
======================
40
```

The initial board state includes several Grass entities (.), Sheep (S) and 2 Wolves (W) in the corners. After the board state is defined, the number of steps to simulate is provided. In this case, the program will simulate 40 steps before printing out the resulting board:

```
======================
|      W  ...........|
|  SS     ...........|
|         ...........|
|  S     ...........|
|  S S   ...........|
|S S      ...........|
|         ...........|
|.  S S    ..........|
|.S        W ........|
|..SSS       S.......|
======================
```

Classes overview

This section describes the classes and their expected functionalities in detail.

Grid

```
class Grid {
    private:
        Entity* cell[BOARD_SIZE_W][BOARD_SIZE_H];

    public:
        Grid(): cell() {}
        ~Grid();

        // Returns true if x and y results in out of bound position.
        bool outOfBounds(int x, int y) const;

        // Get the entity at position x, y. Returns nullptr if position is
        Entity* getCell(int x, int y) const;

        // Set the cell at position x, y to point to entity, and update en
        void setCell(Entity* entity, int x, int y);

        // Delete the cell at position x, y and set to nullptr.
        void deleteCell(int x, int y);
};
```

This class is defined in the `board.h` file. It simply holds a 2D array of Entity pointers, representing a grid with size `BOARD_SIZE_W` x `BOARD_SIZE_H` (20x10).

The 2D pointer array is initialized to be all `nullptr`, and the destructor will call `delete` on all Entity pointers. This is because the grids for different time steps will hold pointers pointing to completely different Entities, so grids holding "old" board states will need to delete all "old" Entities. When the board calls `update()` on all Entities, a new Grid representing the next board state will be passed into the function, and Entities need to add a new copy of themselves or clones onto the next Grid.

Additional functions are provided to help with modifying the Grid:

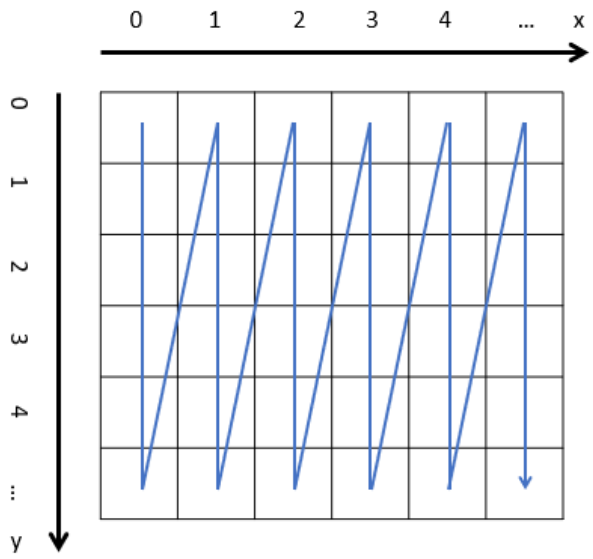
- `getCell()` returns the Entity pointer at **column x, row y** (from the top-left). If the position is out of bounds or there are no Entities, the function returns `nullptr`.
- `setCell()` updates the cell at **column x, row y** to the passed Entity pointer. If the position is out of bounds, **the Entity is deleted**; otherwise, the current cell is deleted (if any), and the cell is assigned to point at the new entity. Additionally, the function **sets the Entity's x and y to the assigned position**. This ensures any Entity not held by the current Grid will be deleted.
- `deleteCell()` simply deletes the Entity at position x, y and sets the pointer to `nullptr`.

Coordinate system for Grid.

Board

```
class Board {
    private:
        Grid* grid;
        int curStep;
        int totalEntityCount {0};

    public:
        Board(): grid(new Grid{}), curStep(0) {}
        ~Board() { delete grid; }

        Grid* getGrid() const { return grid; }

        void updateStep();

        void updateSteps(int numSteps);
        void updateTill(int stepToUpdateTo);

        void print() const;

        void addEntityCount(int modifier) { totalEntityCount += modifier;
};
```

The Board class keeps track of the current board state and step number. Function `updateStep()` will update the board state by 1 step by creating a new Grid, calling all Entities' `update()` function to fill the new Grid with new Entities, and finally `delete` the old Grid and assign the pointer to the new Grid. Functions `updateSteps()` and `updateTill()` are simply helpers that call `updateStep()` to instantly update some number of steps.



Order of Entities being updated. Loops through each column from left to right.

*Note: Because we call `update()` on every Entities, it is guaranteed that all Entities will "perform" their update actions, **except if they would be removed by an Entity before it (clarification below)**. For example, a Grass may spread and get eaten in the same step, resulting in 4 Grasses occupying the adjacent cells but no Grass in the original cell. Another example, a Sheep may eat an adjacent Grass and destroy it, breed with an adjacent Sheep and spawn a new Sheep nearby, move onto an adjacent tile with another Grass and overwrite it, and get eaten by an adjacent Wolf all in the same step.*

*Note: The `updateStep()` function calls `update()` on Entities **which are not removed**. This can lead to asymmetric behaviour: if a Sheep eats a Grass on the right, the Grass is `removed` by the time it would have called `update()`, so it might not spread to adjacent tiles; if the Grass was on the left, it calls `update()` first and might spread to adjacent tiles before being eaten - in this case, there would be 4 (or 3) new Grass tiles.*

The `print()` function outputs the current board state and step number to the terminal. To help with debugging, an additional counter that tracks the number of Entity objects made is stored in `totalEntityCount`. If this value is not equal to the actual number of entities on the board (shown in the printed board state), you will likely have some un-deleted Entities that caused a memory leak. Otherwise, you don't need to worry about this variable and `addEntityCount()`.

Entity

```
class Entity {
    private:
        Board* board;
        int x, y;
        int nextX, nextY;
        bool removed;
        Entity* original;

    protected:
        // Put a copy of current Entity onto the next Grid. Also put the c
        virtual void putSelf(Grid* nextGrid, int x, int y) = 0;

        // Put a brand new Entity onto the next Grid.
        virtual void putClone(Grid* nextGrid, int x, int y) const = 0;

    public:
        Entity(Board* board);
        ~Entity();

        // Accessors
        Board* getBoard() const { return board; }
        int getX() const { return x; }
        int getY() const { return y; }
        bool isRemoved() const { return removed; }

        // Mutators
        void setX(const int x) { this->x = x; }
        void setY(const int y) { this->y = y; }

        // Set nextX and nextY to locate this entity on the nextGrid.
        void setNextSelf(Entity* entity);

        // Called when this entity would be killed. Need to remove copy fr
        void removeSelf(Grid* nextGrid);

        virtual void update(Grid* nextGrid) = 0;
        virtual char toChar() const = 0;
};
```

The Entity class is the **abstract base class** inherited by the classes representing different species. It describes the shared data members and expected functionalities of all species.

Data members: An Entity keeps track of the Board, and its x and y coordinate on the board's current Grid. When initialized, only a Board pointer is supplied, and the coordinates are set when the Entity is placed on the Grid (via `setCell()`). Additionally, `removed` is set to true if and only if the Entity would be deleted from the board, and thus should not appear in the next Grid. Because of the pre-determined order of update, it is possible that an Entity is updated, has placed itself on the next Grid, and is deleted by some other Entity. Therefore, it needs to keep track of its copy's position on the next Grid via `nextX` and `nextY` to remove it if needed. At the same time, the copy keeps track of the original Entity via `original`, in case the copy is deleted by itself.

Protected member functions: These pure virtual functions are to be implemented by the derived classes to help place either the Entity's copy or new clones onto the new Grid. `putSelf()` should put a copy of the current Entity (by calling copy constructor) onto `nextGrid` at the specified position, while `putClone()` should create a new Entity object and put it onto `nextGrid`. These functions need to be implemented by the derived classes since there are additional constraints when placing Entities on the Grid.

Public member functions:

- Derived classes need to override the 2 virtual functions: `update()` and `toChar()`. `update()` should modify the passed Grid according to the class's update scheme, while

`toChar()` returns a character representing the species to be printed on the terminal.

- `setNextSelf()` should be called when the Entity's copy is created and placed on the next Grid, so that the copy can be deleted if this Entity is removed from the board. Simply put, `putSelf()` should call this function. This will update `nextX`, `nextY`, and `original` accordingly.
- `removeSelf()` should be called when this Entity is removed from the board (e.g. being eaten by another Entity). This updates the `removed` variable and, if `nextX` and `nextY` have been modified, delete the copy from the next Grid. To make sure that we are actually deleting the copy and not another Entity that took its position, we only delete if that Entity's `original` points to this Entity.

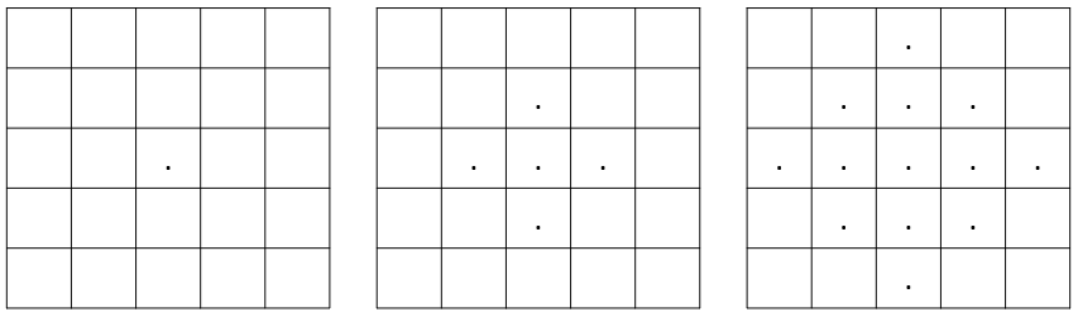
  *Note: To clarify, an Entity is `removed` if it would be destroyed **during** step update, which only happens when that Entity is eaten. We consider Animal dying of hunger to be destroyed **after** step update, so you don't need to call `removeSelf()` when an Animal dies of hunger. In this case, an Animal could be dying of hunger and getting eaten in the same step. You also don't need to worry about Grass being overwritten, as we only overwrite the new Grass pointer in this case.*

Grass

```
class Grass: public Entity {
    private:
        int spreadCounter;

    protected:
        const int getSpreadCooldown() const { return SPREAD_COOLDOWN; }

        virtual void putSelf(Grid* nextGrid, int x, int y) override;
        virtual void putClone(Grid* nextGrid, int x, int y) const override

    public:
        Grass(Board* board): Entity(board), spreadCounter(SPREAD_COOLDOWN)
        Grass(const Grass& grass): Entity(grass.getBoard()), spreadCounter

        virtual void update(Grid* nextGrid) override;
        virtual char toChar() const override { return '.'; }
};
```

The Grass class represents grass tiles on the board. The behaviour of Grass:

- Every `SPREAD_COOLDOWN` step, "spread" to adjacent tiles. This means spawning 4 Grass clones on the orthogonally adjacent tiles. Grass can only spread onto tiles not already occupied by another Entity.
- Otherwise, the Grass stays at its position until it is eaten or overwritten by another Entity moving on top of its place.



Initial state | After SPREAD_COOLDOWN steps | After 2*SPREAD_COOLDOWN steps

The function `toChar()` is already implemented - Grass is represented on the board with a dot ("."). The above image illustrates how the Grass spreads to adjacent tiles.

You will need to implement the virtual functions `putSelf()`, `putClone()` and `update()` according to the described functionalities of each function in the description of class Entity. In **grass.cpp**, a helper function

```
bool putGrass(Grass* grass, Grid* nextGrid, const int x, const int y)
```

will need to be implemented, which should place a Grass entity on the next Grid.

Animal

```
class Animal: public Entity {
    private:
        int hungerCounter;
        int breedCounter;
        int moveCounter;

    protected:
        virtual const int getHungerCooldown() const = 0;
        virtual const int getBreedCooldown() const = 0;
        virtual const int getMoveCooldown() const = 0;

        bool putAnimal(Animal* animal, Grid* nextGrid, int x, int y) const

        void setHungerCounter(const int hungerCounter) { this->hungerCount
        void setBreedCounter(const int breedCounter) { this->breedCounter

        int getRandomMovementIndex(Grid* nextGrid) const;

    public:
        Animal(Board* board, int hungerCounter, int breedCounter, int move
            Entity(board), hungerCounter(hungerCounter), breedCounter(bree

        int getHungerCounter() const { return hungerCounter; }
        int getBreedCounter() const { return breedCounter; }
        int getMoveCounter() const { return moveCounter; }

        virtual void update(Grid* nextGrid) override;

        virtual void eat(Grid* nextGrid) = 0;
        virtual void breed(Grid* nextGrid) = 0;
        virtual void move(Grid* nextGrid) = 0;
};
```

The Animal class is another **abstract base class**, defining the shared properties of Sheep and Wolf. It describes the general behaviour of an Animal:

- In each step, the Animal needs to perform the action "eat". The hunger counter goes down by 1, and if this reaches 0, the animal dies. Otherwise, the Animal tries to eat a suitable Entity on an adjacent tile, depending on the type of Animal. If successful, that Entity is destroyed (`removed`) and the Animal's hunger goes back to maximum value, which is defined by the derived class (`[class]_HUNGER_COOLDOWN`).
- Then, the Animal tries to breed if there is an Animal of the same type on any of the 8 adjacent tiles. This also requires the Animal to have sufficient hunger (more than 70% of maximum value). If successful, it spawns a new Animal of its type on one of the 8 adjacent tiles, which is not occupied by another Animal, chosen randomly. This is independent of the other Animal's hunger status, and both Animals can breed in the same step. An animal can only breed once in every number of steps defined by the derived class (`[class]_BREED_COOLDOWN`).
- Finally, the animal moves to one of the 8 adjacent tiles, also after the number of steps defined by the derived class (`[class]_MOVE_COOLDOWN`). It can only move onto a tile not occupied by another Animal; if no such adjacent tile exists, the animal stays at its position. The movement scheme depends on the derived class.

Therefore, class Animal can implement the `update()` function to follow the above 3-stage scheme for every animal. On the other hand, 3 new pure virtual functions are added for the `eat()`, `breed()` and `move()` stages, which the derived classes will need to implement so that the next Grid can be updated accordingly.

In addition, the function `putAnimal()` is provided to help place Animals on the Grid, similar to the `putGrass()` function in **grass.cpp**. Unlike Grass which cannot overwrite any other Entity, Animal can be placed onto a tile with Grass, overwriting and deleting it. The function's visibility is `protected`, so that derived classes can make use of this function.

The function `getRandomMovementIndex()` returns a number corresponding to one of the 8 adjacent tiles of the Animal that can be used for movement (does not contain an Animal). You should use this function whenever an adjacent tile without an Animal is needed, either for movement or for spawning a new Animal after breeding. The random function is defined in **helper.cpp**, ensuring that Entity behaviours are consistent across program runs. More clarification on how to use the returned number (*adjacency index*) to get the cell position will be described in the **helper.cpp/.h** section.

Sheep

```
class Sheep: public Animal {

    protected:
        virtual const int getHungerCooldown() const override { return SHEE
        virtual const int getBreedCooldown() const override { return SHEEP_
        virtual const int getMoveCooldown() const override { return SHEEP_

        virtual void putSelf(Grid* nextGrid, int x, int y) override;
        virtual void putClone(Grid* nextGrid, int x, int y) const override

    public:
        Sheep(Board* board):
            Animal(board, SHEEP_HUNGER_COOLDOWN / 2, SHEEP_BREED_COOLDOWN,
        Sheep(const Sheep& sheep):
            Animal(sheep.getBoard(), sheep.getHungerCounter(), sheep.getBr

        virtual void eat(Grid* nextGrid) override;
        virtual void breed(Grid* nextGrid) override;
        virtual void move(Grid* nextGrid) override;

        virtual char toChar() const override { return 'S'; }
};
```

Class Sheep is a concrete implementation of class Animal. The virtual functions to get the cooldown for each stage are implemented with the Sheep-specific constants. A Sheep can be created using the normal constructor, which initializes the hunger counter at 50% maximum value and the rest at full value. Alternatively, the copy constructor can be used to create a copy of a Sheep to be placed on the next Grid. Sheep are represented on the board using the letter "S".

Since this class describes a concrete species, the functions `putSelf()`, `putClone()`, `eat()`, `breed()` and `move()` will need to be implemented in **sheep.cpp**.

`eat()`: This function is called every step. The Sheep should check the 8 adjacent tiles of the current Grid to see if any of them are of type Grass. If a Grass is found, destroy it (by calling `removeSelf()`) and reset the Sheep's hunger to full. If there are more than 1 adjacent Grasses, the Sheep should only destroy the first Grass counting from the top left, down each column. In other words, destroy the Grass with the smallest *adjacency index* (see **helper.cpp/.h** section).

`breed()`: This function is called every step after `SHEEP_BREED_COOLDOWN` step are reached after breeding. Similar to `eat()`, if any adjacent Sheep is detected, spawn a new Sheep on a random adjacent tile not occupied by another Animal.

`move()`: This is called every `SHEEP_MOVE_COOLDOWN` step. A Sheep simply moves onto a random adjacent tile not occupied by another Animal and stays in the same position if it cannot move.

Wolf

```
class Wolf: public Animal {

    protected:
        virtual const int getHungerCooldown() const override { return WOLF_
        virtual const int getBreedCooldown() const override { return WOLF_
        virtual const int getMoveCooldown() const override { return WOLF_M

        virtual void putSelf(Grid* nextGrid, int x, int y) override;
        virtual void putClone(Grid* nextGrid, int x, int y) const override

    public:
        Wolf(Board* board):
            Animal(board, WOLF_HUNGER_COOLDOWN / 2, WOLF_BREED_COOLDOWN, W
        Wolf(const Wolf& wolf):
            Animal(wolf.getBoard(), wolf.getHungerCounter(), wolf.getBreed

        virtual void eat(Grid* nextGrid) override;
        virtual void breed(Grid* nextGrid) override;
        virtual void move(Grid* nextGrid) override;

        virtual char toChar() const override { return 'W'; }
};
```

Class Wolf is a concrete implementation of Animal. It is similar to Sheep but only differs in the implementation of the update functions. Wolf is represented on the board using the letter "W".

`eat()`: This is similar to the implementation of Sheep, except that Wolf eats Sheep instead of Grass.

`breed()`: This is similar to the implementation of Sheep, except that Wolf breeds by finding an adjacent Wolf.

`move()`: The Wolf chases after Sheep to eat them. Each Wolf scans the entire current Grid and finds the positions of all Sheep on the board, keeping track of their distances from the Wolf. Once the nearest Sheep has been found, the Wolf moves to the adjacent tile closest to it. If this tile is occupied, the Wolf stays in its place. For consistency, if there are multiple Sheeps with the same distance, it should chase after the first Sheep from the top left, down each column (same as the order of `update()`). If there are no Sheep on the board, the Wolf moves randomly like a Sheep.

helper.cpp/.h

These files define helper functions to make writing the class implementations easier.

```
bool countdown(int& counter, const int MAX_COUNT);
```

Subtracts 1 from the `counter` variable (passed by reference). If it reaches 0, it is reset to `MAX_COUNT`, and the function returns true (otherwise false). This function can be used to execute some code only every specific number of steps:

```
bool countdownReached = countdown(counter, COOLDOWN);
if (countdownReached) {
    // code that is executed once every COOLDOWN steps
}
else {
    // code to be executed during "cooldown" steps
}
```

Note: You should do countdown on all counter variables before creating Entity copies with the copy constructor since the copied Entity should represent the original Entity in the next step.

```
unsigned int pseudo_rand();
```

This function returns a random integer from 0 to 32767. You don't need to use this function; it is only used in `Animal::getRandomMovementIndex()` which is already given.

```
int getAdjX(const int index);
int getAdjY(const int index);
```

Returns the x, y offset from the passed *adjacency index*. We assign the following index for each of the 8 adjacent tiles:

0	3	6
1		7
2	5	8

The adjacency index is returned from the `Animal::getRandomMovementIndex()` function. To get the cell position from the index, you can use the above 2 functions:

```
int adjX = getX() + getAdjX(index);
int adjY = getY() + getAdjY(index);
```

End of Description

Tasks

Task 1: Implement Grass

For this task, you need to implement the following functions in **grass.cpp**, as specified in the assignment introduction:

```
bool putGrass(Grass* grass, Grid* nextGrid, const int x, const int y)
```

This function tries to set `nextGrid`'s cell at position x and y to point to `grass`, and returns a boolean of whether `grass` was placed successfully on `nextGrid`. If the position is out of bounds or the cell is already pointing at another Entity, `grass` should be deleted, and the function should returns false. Otherwise, modify `nextGrid` accordingly and return true.

```
void Grass::putSelf(Grid* nextGrid, const int x, const int y)
```

Place a copy of this Grass onto `nextGrid` at the specified position. You should use the copy constructor to create a Grass object representing this Grass in the next step. You may use `putGrass()` implemented above. If the copy was placed successfully, call `setNextSelf()` to properly link this Grass with the copy for deletion if necessary.

```
void Grass::putClone(Grid* nextGrid, const int x, const int y) const
```

Place a brand new Grass onto `nextGrid` at the specified position. You should use the normal constructor to create a new Grass object. You may use `putGrass()` implemented above.

```
void Grass::update(Grid* nextGrid)
```

Update the `nextGrid` according to the update rules of Grass. If the spread countdown is reached, spawn 4 Grasses onto the orthogonally adjacent tiles using `putClone()`. You may find the `countdown()` function in **helper.cpp** useful. Otherwise, put a copy of this Grass onto this position using `putSelf()`. You don't need to worry about checking if the current cell has another Entity; it is done in `putGrass()`.

Task 2: Implement Animal

For this task, you need to implement the following functions in **animal.cpp**:

```
bool Animal::putAnimal(Animal* animal, Grid* nextGrid, const int x, const
```

This function tries to set `nextGrid`'s cell at position x and y to point to `animal`, and returns a boolean of whether the `animal` was placed successfully on `nextGrid`.

You can implement this similar to `putGrass()` in Task 1. However, if the current cell in `nextGrid` points to Grass, allow the animal to be placed there. Otherwise, if the current cell contains an Animal, delete the passed `animal` and return false.

Note: You don't need to worry about deleting the original Grass. Here you overwrite and delete a Grass pointer on the next Grid, which can either be a Grass copy of a current Grass tile or new clones made from Grass's "spread". If the original Grass is deleted via `removeSelf()`, it will check if the Entity on the next Grid's specified position is indeed its copy (via `original`), so that we avoid accidentally deleting a different Entity.

Note: Calling `deleteCell()` when overwriting a Grass tile is optional, as `setCell()` will also delete the current Entity.

```
void Animal::update(Grid* nextGrid)
```

Implement the 3-stage update scheme of Animal. The code has been partially implemented to help you ensure functions are called at the right steps, so that the output is consistent with our test cases.

First, countdown the `hungerCounter`. If it reaches 0, the animal dies, and the function can simply return. Otherwise, call `eat()` to update the grid and hunger.

Note: If the animal dies of hunger, you do not need to call `removeSelf()`.

Next, if `breedCounter` is 1 and `hungerCounter` is more than 70% of the maximum value, the animal tries to breed by calling `breed()`. If `breedCounter` is greater than 1, do countdown instead.

Note: Floating point comparison may behave differently on different platforms. In the given demo program, you may find that `breed()` is still called when hunger value is exactly 70%. As long as you don't modify the code already implemented to check for hunger condition, there should be no problem as all submissions will be graded on ZINC, along with the solution code to cross-check.

Finally, countdown the `moveCounter`. If it reaches 0, call `move()` to move the animal. Otherwise, put the animal's copy in its position.

Task 3: Implement Sheep

For this task, you need to implement the following functions in **sheep.cpp**:

```
void Sheep::putSelf(Grid* nextGrid, const int x, const int y)
void Sheep::putClone(Grid* nextGrid, const int x, const int y) const
```

Implement these functions similar to the Grass functions in Task 1. You can make use of `putAnimal()` implemented in Task 2.

```
void Sheep::eat(Grid* nextGrid)
```

Check if any adjacent cell of the current Grid points to Grass. If a Grass is found and has not been removed, destroy it using `removeSelf()`, and reset the `hungerCounter` to full. A Sheep can only eat at most once per turn.

```
void Sheep::breed(Grid* nextGrid)
```

Check if any adjacent cell of the current Grid points to Sheep. If a Sheep is found, try to spawn a new Sheep on an adjacent cell. You can use the `getRandomMovementIndex()` function to get an available adjacent cell. If a cell is found, spawn a new Sheep at that position and reset the `breedCounter` to full. A Sheep can only breed at most once per turn.

Note: No need to check if adjacent Sheep is `removed`.

```
void Sheep::move(Grid* nextGrid)
```

Get a random available adjacent cell to move to using `getRandomMovementIndex()`. If there is an adjacent cell, place the copy of this Sheep there. Otherwise, place it in its current position.

Task 4: Implement Wolf

For this task, you need to implement the following functions in **wolf.cpp**:

```
void Wolf::putSelf(Grid* nextGrid, const int x, const int y)
void Wolf::putClone(Grid* nextGrid, const int x, const int y) const
```

Implement these functions similar to the Sheep functions.

```
void Wolf::eat(Grid* nextGrid)
void Wolf::breed(Grid* nextGrid)
```

Implement these functions similar to the Sheep functions. Keep in mind that Wolf eats Sheep and breeds with Wolf.

```
void Wolf::move(Grid* nextGrid)
```

If there is at least one Sheep on the board, Wolf tries to move towards it. You should loop over the board in the same order as Board calling `update()`, calculate Euclidean distance to each cell containing a Sheep, and keep track of the shortest distance found. Ignore Sheep that have been eaten (`removed`), and no need to keep track of Sheep with the same distance as the shortest one found so far. If a Sheep is found, move to the adjacent cell closest to that Sheep. If **that cell is occupied with another Animal**, the Wolf stays in its position. If no Sheep is found, the Wolf moves randomly like a Sheep.

Note: Sheep that died of hunger can still be chased after, since they are not `removed`.

*Note: When checking if **cell is occupied with another Animal**, you need to check **both** the current Grid and next Grid. If current Grid has an Animal there, Wolf should not move onto that position to not overwrite the animal. If next Grid has an Animal there, that means some other Animal has moved there and Wolf cannot go there anymore.*

(This is the behaviour in `getRandomMovementIndex()`, which only selects from adjacent cells that are not occupied in both the current and next Grids).

End of Tasks

Testing

The test cases are provided in the **inputs** and **outputs** folder. **outputs/output{}.txt** is the expected output when supplying **inputs/input{}.txt** as input to the program. Compared to the example **input.txt**, these are mostly simpler so that you can find test cases related to some specific function and debug.

List of provided test cases and their expected program behaviour:

1. Single Grass entity, expects to spread 3 times.
2. 2 Grass entities, expects to spread 5 times and overlap.

3. Single Sheep entity, expects to move to an adjacent tile.
4. Single Sheep and 2 Grasses, expects Sheep to eat the top left Grass.
5. Single Sheep and 2 Grasses, expects Sheep to starve after 18 steps.
6. 2 Sheep and multiple Grasses, expects Sheep to breed.
7. 4 Wolves, expects to move randomly.
8. 4 Wolves and 1 Sheep, expects Wolves to move towards Sheep.
9. 1 Wolf and 2 Sheep, expects to move towards the leftmost Sheep.
10. 1 Wolf and multiple Grasses, expects to move onto and overwrite Grass.

During grading, we will use different test cases that will cover a wider range of scenarios. Your score will be determined based on the number of tests passed. There will be additional points given if your program does not have memory leak for each test case.

End of Testing

Resources & Sample Program

- Skeleton code: source.zip **(Last updated 22:20 21/10/2022)**
- Demo program (executable for Windows): PA2.exe **(Last updated 03:10 29/10/2022)**
- Demo debug program (executable for Windows): PA2_debug.exe **(Last updated 03:10 29/10/2022)**
- Demo program (executable for Mac): PA2_mac.exe **(Last updated 20:30 01/11/2022)**
- Demo debug program (executable for Mac): PA2_debug_mac.exe **(Last updated 20:30 01/11/2022)**

The debug program is the same as the demo program, but it also prints messages logging Sheep and Wolf behaviour in each step. This will hopefully help you compare your program output and behaviour with the demo program.

Types of logged messages: Sheep/Wolf eating Grass/Wolf, Sheep/Wolf breeding and spawning new clone, Sheep/Wolf moving to a new cell, and Sheep/Wolf dying of hunger. You can use command line arguments to disable some types of messages:

```
.\PA2_debug.exe
```

Runs the debug program and prints everything.

```
.\PA2_debug.exe -S e -W b
```

Runs the debug program and prints everything except Sheep **e**ating and Wolf **b**reeding.

```
.\PA2_debug.exe -S em -W bs
```

Runs the debug program and prints everything except Sheep **e**ating and **m**oving, and Wolf **b**reeding and **s**tarving. (Note that any order is fine)

```
.\PA2_debug.exe -S ebms -W ebms
```

Runs the debug program and prints nothing (except step number).

End of Download

Submission and Deadline

Deadline: 23:59:00 on November 12, 2022.

ZINC Submission

Please submit the following files to ZINC by zipping the following 4 files. ZINC usage instructions can be found here.

```
grass.cpp
animal.cpp
sheep.cpp
wolf.cpp
```

Notes:

- You may submit your file multiple times, but only the latest version will be graded.
- Submit early to avoid any last-minute problems. Only ZINC submissions will be accepted.
- The ZINC server will be very busy on the last day, especially in the last few hours, so you should expect you would get the grading result report not-very-quickly. However, as long as your submission is successful, we will grade your latest submission with all test cases after the deadline.
- If you have encountered any server-side problems or webpage glitches with ZINC, you may post on the ZINC support forum to get attention and help from the ZINC team quickly and directly. If you post on Piazza, you may not get the fastest response as we need to forward your report to them, and then forward their reply to you, etc.

Compilation Requirement

It is **required** that your submissions can be compiled and run successfully in our online autograder ZINC. If we cannot even compile your work, it won't be graded. Therefore, for parts you cannot finish, just put in a dummy implementation so that your whole program can be compiled for ZINC to grade the other parts you have done. Empty implementations can be like:

```
int SomeClass::SomeFunctionICannotFinishRightNow()
{
    return 0;
}

void SomeClass::SomeFunctionICannotFinishRightNowButIWantOtherPartsGraded(
{
}
```

Reminders

Make sure you actually upload the correct version of your source files - we only grade what you upload. Some students in the past submitted an empty file or a wrong file or an exe file which is worth zero mark. So **you must double-check the file you have submitted**.

Late Submission Policy

There will be a penalty of -1 point (out of a maximum 100 points) for every minute you are late. For instance, since the deadline for assignment 2 is 23:59:00 on Nov 12th, if you submit your solution at 1:00:00 on Nov 13th, there will be a penalty of -61 points for your assignment. However, the lowest grade you may get from an assignment is zero: any negative score after the deduction due to late penalty (and any other penalties) will be reset to zero.

End of Submission and Deadline

Frequently Asked Questions

Q: My code doesn't work / there is an error. Here is the code. Can you help me fix it?
A: As the assignment is a major course assessment, to be fair, you are supposed to work on it on your own, and we should not finish the tasks for you. We might provide some very general hints, but we shall not fix the problem or debug it for you.

Q: Can I add extra helper functions?
A: You may do so in the files that you are allowed to modify and submit. That implies you cannot add new member functions to any given class.

Q: Can I include additional libraries?
A: No. Everything you need is already included - there is no need for you to add any include statement (under our official environment).

Q: Can I use global or static variables such as `static int x`?
A: No.

Q: Can I use `auto`?
A: No.

End of Frequently Asked Questions

Change Log

22:30 - 21/10/2022: Updated the source files:

- **entity.h**: Made the destructor `~Entity()` virtual.
- **main.cpp**: Fixed input parse which may cause buffer overflow.

12:20 - 22/10/2022: Updated assignment deadline.

22:10 - 22/10/2022: Added some clarifications to the description. Changes are highlighted in **red**.

19:15 - 23/10/2022: Clarified `eat()` behaviour - if `hungerCounter` was 1, it counts down to 0 and the animal dies; otherwise, call `eat()`.

19:15 - 26/10/2022: Added a demo program with Animal activity output for debug purposes.

03:10 - 29/10/2022: Clarified `Wolf::move()` behaviour for a specific case and updated the demo programs.

18:50 - 31/10/2022: Clarified the asymmetric behaviour due to `updateStep()` ignoring `removed` Entities. No change to code or task implementations are needed.

20:30 - 01/11/2022: Added demo programs for Mac.

15:30 - 12/11/2022: Clarified `Sheep::breed()` that `removed` Sheeps can still be used as breed target. Also clarified the floating point comparison issue regarding breed's 70% hunger requirement.

End of Frequently Asked Questions