OOP project 1. C++
Dawid Bach 184629 ID II semester

9.05.2021

Plan:

- World class and its visualization
  1. Fields elaborating
  2. Constructors
  3. Destructor
  4. makeTurn() function
  5. drawWorld() function
  6. game() function

- Organism
  1.Brief summary

- Animals
  1. Action() function
  2. Collision() function
  3. Breeding
  4. Default subclass constructor
  5. Fox's Action() function
  6. Turtle's Action() and Collision() function
  7. Antelope's Action() and Collision() function

- Plants
  1.Action() function
  2.Collision() function
  3. Sowing
  4. Sow thisle's Action() function
  5. Guarana's Collision() function
  6. Belladonna's Collision() function
  7. Sosnowky's hogweed Action() and Collision() function

- Human
  1.Initialize player
  2.Special ability

- Loading and saving
  1.File format
  2.Loading
  3.Saving

World class and its visualization

```
class World {
private:
    int m;
    int n;
    int special_ability_cnt;

    Species toSpecies(const char t);
    void newOrganism(Species sp, const int cell);
    int loadGame();
    void saveGame();
    void makeTurn(Commands dir);
    void drawWorld();
public:
    bool failed_to_load;
    bool player_alive;
    std::vector<Organism*> organisms;
    Cell* cells;
    std::string comment;

    World();
    World(const int n, const int m);
    void resetAbilityCnt();
    int getAbilityCnt() const;
    int getM() const;
    int getN() const;
    void game();
    ~World();
};
```

1. **Fields elaborating**

-**int m** and **int n** stand for the size of game's grid (n = y axis, m = x axis)
-**int special_ability_cnt** is used to store current cooldown of Human's special ability
-**bool failed_to_load** is a flag used to recognize if everything during loading save went fine
-**bool player_alive** is an information about Human's state
-**Organisms** are stored in vector of pointers of Organism type

- **Cells** is 1-dimensional array of every cell on map

```
struct Cell {
    Position position;
    Organism* occupant;
    bool isOccupied;
};
```

It holds pointer to its occupant, position (picture below) and if it's occupied.

```
struct Position {
    int x;
    int y;

    bool operator==(const Position& right)
    {
        if ((x == right.x) && (y == right.y)) return true;
        else return false;
    }

    std::string toString()
    {
        return "(" + std::to_string(x) + "," + std::to_string(y) + ")";
    }

    friend std::ostream& operator<<(std::ostream& ostr, const Position& pos)
    {
        ostr << "(" << pos.x << "," << pos.y << ")";
        return ostr;
    }
};
```

This is the form in which position of every organism/cell will be stored.

- **Comment** is used to store all the events during turn

2. **Constructors**
   New game constructor:

```
World::World(int n, int m)
    :n(n), m(m)
{
    comment = "";
    failed_to_load = false;

    cells = new Cell[n * m];
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            cells[(m * i) + j].position.x = j;
            cells[(m * i) + j].position.y = i;
            cells[(m * i) + j].occupant = NULL;
            cells[(m * i) + j].isOccupied = false;
        }
    }

    player_alive = true;
    special_ability_cnt = 0;
    organisms.push_back(new Human(n, m));
    const int human_pos = organisms[0]->position.y * m + organisms[0]->position.x;
    cells[human_pos].isOccupied = true;
    cells[human_pos].occupant = organisms[0];

    int organisms_number = sqrt(m * n) * 3;
    if ((m * n) == 1) organisms_number = 1;
    else if (organisms_number >= (m * n)) organisms_number /= 2;

    if (n * m < ORGANISMS_LOWER_LIMIT)
    {
        for (int i = 1; i < organisms_number; i++)
        {
            Species rand_species = (Species)(rand() % 11);
            newOrganism(rand_species, -1);
        }
    }
    else
    {
        for (int i = 0; i < 11; i++)
        {
            for (int j = 0; j < 2; j++)
                newOrganism((Species)i, -1);
        }

        const int diff = organisms_number - 22;
        for (int i = 0; i < diff; i++)
        {
            Species rand_species = (Species)(rand() % 11);
            newOrganism(rand_species, -1);
        }
    }
}
```

Firstly the cells array is initialized with default values and the player is created and setted. Then I'm calculating the numbers of organism that should be initially placed on the map. I came up with some random formula that will always keep it below number of cells, but also make this number reasonable.
In order to perform normal game, when we can see at least **2 representants** from each species, the total number of cells (**n*m**) must be bigger than **50**. It's because there are 11 types, which with the player already gives us **23 organisms**. I've bounded it to 50 to leave some space for their movements. If there are more organisms that should be spawned it is filled randomly.

If this condition is not satisfied, organisms will be spawned randomly.

Loaded game constructor:

```
World::World()
{
    system("cls");
    failed_to_load = false;
    if (loadGame() == -1) failed_to_load = true;
}
```

I will report the loadGame() function in the **Load and saving** section, just to make it consistent.
For now it creates the **world** with settings from a loaded file, and if something goes wrong it sets **failed_to_load** flag to true.

3. Destructor

```
World::~World()
{
    if (cells != NULL) delete cells;
    std::vector<Organism*>().swap(organisms);
}
```

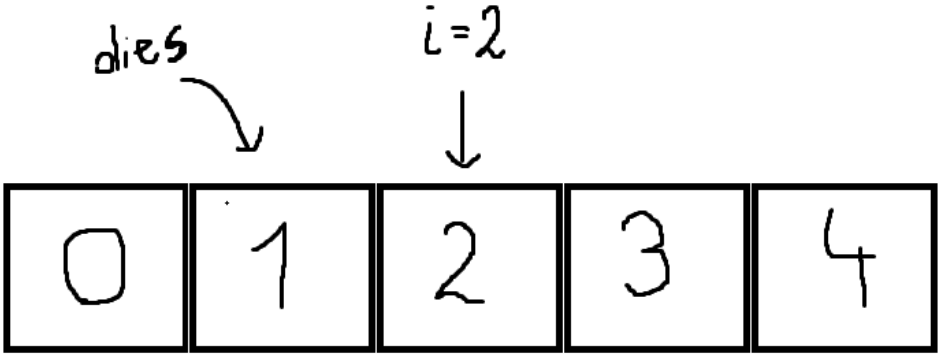
Just freeing **cells** array and **organisms** vector.

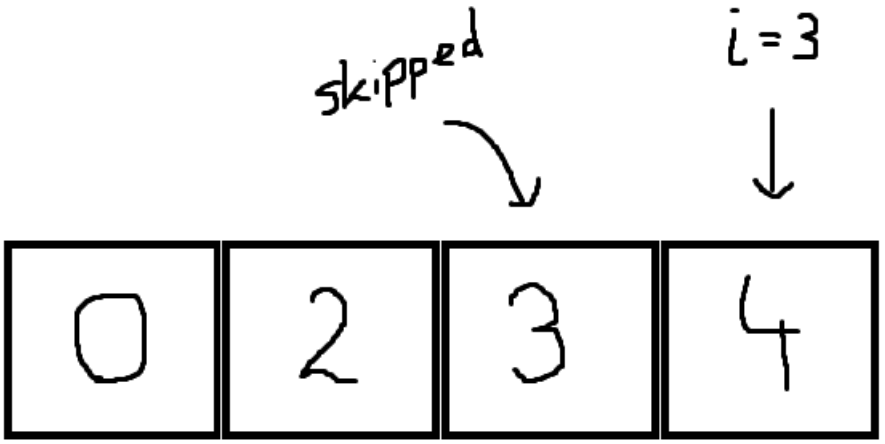
4. makeTurn() function

```
void World::makeTurn(Commands dir)
{
    Sort(organisms, 0, organisms.size() - 1);
    for (int i = 0; i < organisms.size(); i++) {
        organisms[i]->allowed_to_move = true;
    }
    int display_cooldown = special_ability_cnt;
    if (player_alive && special_ability_cnt > 0)
    {
        special_ability_cnt--;
        display_cooldown = special_ability_cnt;
    }
    else if (player_alive && special_ability_cnt == 0 && dir == SPECIAL_ABILITY)
        display_cooldown = COOLDOWN;

    if (player_alive == true)
    {
        comment = comment
            + "Z - Special ability (Cooldown: " + std::to_string(display_cooldown) + " rounds)\n"
            + "L - Load save\n"
            + "S - Save game\n"
            + "Q - Quit game\n\n";
    }
    else
    {
        comment = comment
            + "Z - Special ability (Player is dead!)\n"
            + "L - Load save\n"
            + "S - Save game\n"
            + "Q - Quit game\n\n";
    }

    int size = organisms.size();
    for (int i = 0; i < size; i++)
    {
        if (i >= organisms.size()) break;
        for (int j = 0; j < size; j++)
        {
            if (j >= organisms.size()) break;
            if (organisms[j]->allowed_to_move == true)
            {
                if (organisms[j]->type == HUMAN)
                    organisms[j]->Action(this, dir);
                else
                    organisms[j]->Action(this, NON);
                break;
            }
        }
    }

    size = organisms.size();
    for (int i = 0; i < size; i++)
    {
        organisms[i]->age++;
    }
```

At the beginning of every turn, firstly organisms are sorted by it's initiative and age with **Sort()** function which is implementation of **quicksort** algorithm.
Then every existing organism is allowed to move by setting their **allowed_to_move** field to true.
After this comes some **Human's** ability logic and default comments.
Next there is a loop performing **action** of every organism, ordered by their initiative and age.
If it finds an organism that is allowed to move, it will perform its action and then it will come back to the beginning of the vector.

It is done this way, because of problems with killing organisms. If it were just iterating through vector once and performing action of every organism, some of them might be skipped because of for example **Sosnowky's hogweed action**. (Situation when i = 2, hogweed kills organisms at i = 1, so every organism after 1st position is moved and because of that some may be skipped).





At the end **age** of every existing organism is increased.

5. **drawTurn() function**

```
void World::drawWorld()
{
    for (int i = 0; i < (n * 2) + 1; i++)
    {
        for (int j = 0; j < (m * 2) + 1; j++)
        {
            if (i % 2 == 0)
            {
                if (j % 2 == 0) std::cout << "-";
                else std::cout << "---";
            }
            else
            {
                if (j % 2 == 0) std::cout << "|";
                else
                {
                    char symbol = ' ';
                    int curr_pos = ((i / 2) * m) + j / 2;
                    if (cells[curr_pos].isOccupied) symbol = cells[curr_pos].occupant->Draw();
                    if (cells[curr_pos].isOccupied && cells[curr_pos].occupant->type == HUMAN)
                        std::cout << " " << BOLDBLUE << symbol << RESET << " ";
                    else std::cout << " " << symbol << " ";
                }
            }
        }
        std::cout << std::endl;
    }
}
```

Output for n = 5 and m = 5:

```
---------------------
|   |   |   | s | y |
---------------------
| c | h |   | h | f |
---------------------
| a | f | f | t |   |
---------------------
|   | c | b |   |   |
---------------------
| c | H | a |   |   |
---------------------
```

I've made some adjustments here to make it look more like a square rather than like a rectangle when **n == m** (in place of cell **'---'** instead of **'-'**). **curr_pos** stands for current cell and is calculated like this because I'm using a **1-D array**.
At every cell place I'm checking if a cell is occupied, then the occupant's **Draw()** function is called (it returns a char, symbol of organism). If **Human** in is an occupant, the color is changed to **BOLDBLUE** to make him more visible.

6. game() function

```
void World::game()
{
    system("cls");
    drawWorld();
    if (player_alive == true)
    {
        comment = comment
            + "Z - Special ability (Cooldown: " + std::to_string(special_ability_cnt) + " rounds)\n"
            + "L - Load save\n"
            + "S - Save game\n"
            + "Q - Quit game\n\n";
    }
    else
    {
        comment = comment
            + "Z - Special ability (Player is dead!)\n"
            + "L - Load save\n"
            + "S - Save game\n"
            + "Q - Quit game\n\n";
    }
    std::cout << comment;
    comment = "";

    Commands dir = NON;
    bool quit = false;
    while (!quit)
    {
        dir = NON;
        if (player_alive == true)
        {
            while (dir == NON)
                dir = ConvertKey();
        }
        else dir = ConvertKey();

        if (dir == QUIT)
        {
            break;
        }
```

```
        else if (dir == SAVE_GAME) saveGame();
        else if (dir == LOAD_SAVE)
        {
            if (loadGame() == -1)
            {
                system("cls");
                std::cout << "Wrong file format! Proceed to menu." << std::endl;
                system("PAUSE");
                failed_to_load = true;
                break;
            }
        }

        system("cls");
        if ((dir != SAVE_GAME) && (dir != LOAD_SAVE)) makeTurn(dir);
        else
        {
            if(dir == SAVE_GAME) comment = comment + "Game has been saved to file!\n";
            else if (dir == LOAD_SAVE) comment = comment + "Game has been loaded from file!\n";
        }

        drawWorld();
        std::cout << comment;
        comment = "";
    }
}
```

game() is a function where all the logic of makings turns and managing situations like **save** or **load** is handled.
Mainly it is a while loop, where firstly I'm taking user input (by **ConvertKey()** function) and then according to what player pressed, a special case is handled.
If a player saves a game, or loads a game, the **makeTurn()** function isn't called.

Organism

```
class Organism {
public:
    Position position;
    Species type;
    int strength;
    int age;
    int initiative;
    bool allowed_to_move;

    Organism() {};
    virtual char Draw() const = 0;
    virtual void Collision(Organism* occupant, Organism* attacker, World* world) {};
    virtual void Action(World* world, Commands dir) = 0;
    ~Organism() {};
};
```

Organism class is **pure virtual**, it does nothing by itself. Besides basic fields like **strength**, **age** or **initiative**, I've added **position** (information about position on map), **type** to store information about species of organism and also **allowed_to_move** to prevent situation that I described in **makeTurn()** section.

Animals

```
class Animal : public Organism {
protected:
    Position Move(Commands dir, World* world);
    char Draw() const override;
    int positionToCell(const Position& pos, World* world);
    int FindFreeNeighbor(World* world, const Organism* org);
    std::string speciesToString(Species sp);
    void Breed(World* world, const int cell, Species kind);
    void Defense(World* world, Organism* attacker);
    void Overtake(World* world, Organism* occupant, Organism* attacker);
public:
    Animal() {};
    void Collision(Organism* occupant, Organism* attacker, World* world) override;
    void Action(World* world, Commands dir) override;
    ~Animal() {};
};
```

1. **Action() function**

```
void Animal::Action(World* world, Commands dir)
{
    this->allowed_to_move = false;
    const Position dest_position = Move(dir, world);

    const int actual_cell = positionToCell(position, world);
    const int destiny_cell = positionToCell(dest_position, world);

    if (actual_cell != destiny_cell)
    {
        if (world->cells[destiny_cell].isOccupied)
        {
            if (this->strength > world->cells[destiny_cell].occupant->strength ||
                world->cells[destiny_cell].occupant->type == BELLADONNA ||
                world->cells[destiny_cell].occupant->type == SOSNOWKYS_HOGWEED)
                world->cells[destiny_cell].occupant->Collision(world->cells[destiny_cell].occupant, this, world);
            else
                Collision(world->cells[destiny_cell].occupant, this, world);
        }
        else
        {
            this->position = dest_position;

            world->cells[actual_cell].isOccupied = false;
            world->cells[actual_cell].occupant = NULL;

            world->cells[destiny_cell].isOccupied = true;
            world->cells[destiny_cell].occupant = this;
        }
    }
}
```

Action() function takes (besides World ref) **Command dir**, which can be used to force movement in certain direction (If non is given it will randomly choose direction).
Move() function changes the given position parameter in a certain direction, also given as a parameter. Movements are handled in a way that it allows organisms to move **diagonally**.

After **destiny_cell** is calculated, I'm checking if an organism wants to change its position. If there it's the same as **actual_cell** it simply does nothing. Else, later I'm checking if **destiny_cell** is occupied by an organism, if so then we have to decide which organism collision will be called.
I've noticed that if an organism has bigger strength it wants to fight, so I applied the logic that the **collision of weaker organism is called** to give it a chance to miss fight. (With some exceptions, like **Belladonna** and **Hogweed**. When those two are included in a collision, always their collision will be called.)
Else if the **destiny_cell** isn't occupied it will just move the animal there.

2. Collision() function

```
void Animal::Collision(Organism* occupant, Organism* attacker, World* world)
{
    world->comment = world->comment + "Collision between occupant: " + speciesToString(occupant->type)
        + " at " + occupant->position.toString() + " with attacker: " + speciesToString(attacker->type)
        + " at " + attacker->position.toString() + "\n";
    if (occupant->type == attacker->type)
    {
        int free_cell = FindFreeNeighbor(world, occupant);

        if (free_cell != NOT_FOUND) Breed(world, free_cell, this->type);
        else
        {
            free_cell = FindFreeNeighbor(world, attacker);

            if (free_cell != NOT_FOUND) Breed(world, free_cell, this->type);
        }
    }
    else
    {
        if (occupant->strength > attacker->strength)
        {
            if (attacker->type == HUMAN) world->player_alive = false;
            Defense(world, attacker);
        }
        else
        {
            if (occupant->type == HUMAN) world->player_alive = false;
            Overtake(world, occupant, attacker);
        }
    }
}
```

Firstly comment about collision is added.
Then I'm checking if the occupant and attacker are of the same type, if so I check if there is any **free cell in neighborhood** of occupant or attacker (if there is no free space, they will not breed).
If they're not of the same type, according to what animal is stronger either **Defense()** function or **Overtake()** function is called.
Defense() function is called when the occupant is stronger, then the attacker is deleted.
Overtake() function is called when the attacker is stronger, then the occupant is deleted and the attacker is moved to the occupant's cell.

3. Breeding

```
void Animal::Breed(World* world, const int cell, Species kind)
{
    const int n = world->getN();
    const int m = world->getM();

    switch (kind)
    {
    case WOLF:
        world->organisms.push_back(new Wolf(n, m));
        break;
    case SHEEP:
        world->organisms.push_back(new Sheep(n, m));
        break;
    case FOX:
        world->organisms.push_back(new Fox(n, m));
        break;
    case TURTLE:
        world->organisms.push_back(new Turtle(n, m));
        break;
    case ANTELOPE:
        world->organisms.push_back(new Antelope(n, m));
        break;
    case CYBER_SHEEP:
        world->organisms.push_back(new Cyber_Sheep(n, m));
        break;
    }

    const int last_index = world->organisms.size() - 1;
    world->organisms[last_index]->position = world->cells[cell].position;
    world->cells[cell].isOccupied = true;
    world->cells[cell].occupant = world->organisms[last_index];

    world->comment = world->comment + "New animal of type: " + speciesToString(type)
        + " at position " + world->cells[cell].occupant->position.toString() + "\n";
}
```

The logic of calling **Breed()** function was pretty much explained in the **Collision() section**.
Breed() function simply creates a new animal and places it in the **world**.
The position of a new animal is provided to function as a parameter. (Found by **FindFreeNeighbor()** function)

```
-------------
| f | f | f |
-------------
| f |   |   |
-------------
|   |   | H |
-------------
Z - Special ability (Cooldown: 0 rounds)
L - Load save
S - Save game
Q - Quit game

Collision between occupant: fox at (1,0) with attacker: fox at (0,0)
New animal of type: fox at position (2,0)
Collision between occupant: fox at (0,0) with attacker: fox at (1,0)
New animal of type: fox at position (0,1)
```

4. **Default subclass constructor**

```
Wolf::Wolf(const int n, const int m)
{
    initiative = 5;
    strength = 9;
    age = 0;
    allowed_to_move = false;
    position.x = (rand() % m);
    position.y = (rand() % n);
    type = WOLF;
}
```

Simply basic values are assigned and position is randomized. It's mainly because of initial organisms creating. If an organism is spawned in an already taken cell, it's handled in a way that it will randomize till it finds a free cell. (In world constructor)

5. **Fox's Action() function**

```
void Fox::Action(World* world, Commands dir)
{
    this->allowed_to_move = false;
    const int actual_cell = positionToCell(position, world); //pos_before
    const int neighbor_cell = FindWeakerNeighbor(world, this);

    if (neighbor_cell != NOT_FOUND)
    {
        if (world->cells[neighbor_cell].isOccupied)
        {
            if (this->strength > world->cells[neighbor_cell].occupant->strength ||
                world->cells[neighbor_cell].occupant->type == BELLADONNA ||
                world->cells[neighbor_cell].occupant->type == SOSNOWKYS_HOGWEED)
                world->cells[neighbor_cell].occupant->Collision(world->cells[neighbor_cell].occupant, this, world);
            else
                Collision(world->cells[neighbor_cell].occupant, this, world);
        }
        else
        {
            world->cells[actual_cell].isOccupied = false;
            world->cells[actual_cell].occupant = NULL;

            world->cells[neighbor_cell].isOccupied = true;
            world->cells[neighbor_cell].occupant = this;
            this->position = world->cells[neighbor_cell].position;
        }
    }
}
```

First I check with **FindWeakerNeighbor()** if there is any free or weaker organism in the neighborhood. If so, the **default animal Action()** function is called, if there is no such cell it simply stay in the actual cell and hopes to stay alive.

6. Turtle's Action() and Collision() function

```
void Turtle::Action(World* world, Commands dir)
{
    this->allowed_to_move = false;

    int rand_number = (rand() % 100);
    if (rand_number <= CHANCE_TO_MOVE)
    {
        this->allowed_to_move = false;
        const Position dest_position = Move(dir, world);

        const int actual_cell = positionToCell(position, world);
        const int destiny_cell = positionToCell(dest_position, world);

        if (actual_cell != destiny_cell)
        {
            if (world->cells[destiny_cell].isOccupied)
            {
                if (this->strength > world->cells[destiny_cell].occupant->strength ||
                    world->cells[destiny_cell].occupant->type == BELLADONNA ||
                    world->cells[destiny_cell].occupant->type == SOSNOWKYS_HOGWEED)
                    world->cells[destiny_cell].occupant->Collision(world->cells[destiny_cell].occupant, this, world);
                else
                    Collision(world->cells[destiny_cell].occupant, this, world);
            }
            else
            {
                this->position = dest_position;

                world->cells[actual_cell].isOccupied = false;
                world->cells[actual_cell].occupant = NULL;

                world->cells[destiny_cell].isOccupied = true;
                world->cells[destiny_cell].occupant = this;
            }
        }
    }
}
```

Simply gets some random number in interval 0 - 99, if this number fits under **CHANCE_TO_MOVE** default **animal Action()** is called.

```
void Turtle::Collision(Organism* occupant, Organism* attacker, World* world)
{
    world->comment = world->comment + "Collision between occupant: " + speciesToString(occupant->type)
        + " at " + occupant->position.toString() + " with attacker: " + speciesToString(attacker->type)
        + " at " + attacker->position.toString() + "\n";

    if (attacker->type == occupant->type)
    {
        int free_cell = FindFreeNeighbor(world, occupant);

        if (free_cell != NOT_FOUND) Breed(world, free_cell, this->type);
        else
        {
            free_cell = FindFreeNeighbor(world, attacker);

            if (free_cell != NOT_FOUND) Breed(world, free_cell, this->type);
        }
    }
```

The first part is the same as animal Collision() function (Breeding).

```
else
{
    if (this == occupant)
    {
        if (occupant->strength > attacker->strength)
        {
            if (attacker->type == HUMAN) world->player_alive = false;
            Defense(world, attacker);
        }
        else if (attacker->strength < STRENGTH_TO_REFLECT) world->comment = world->comment + "Defends!\n";
        else
        {
            if (occupant->type == HUMAN) world->player_alive = false;
            Overtake(world, occupant, attacker);
        }
    }
    else if (this == attacker)
    {
        if (occupant->strength < attacker->strength)
        {
            if (occupant->type == HUMAN) world->player_alive = false;
            Overtake(world, occupant, attacker);
        }
        else if (occupant->strength < STRENGTH_TO_REFLECT) world->comment = world->comment + "Turtle reflects attack!\n";

        else if (occupant->strength > attacker->strength)
        {
            if (attacker->type == HUMAN) world->player_alive = false;
            Defense(world, attacker);
        }
    }

}
```

Turtle may reflect attack in both situations, when it's the attacker and when it's the occupant.

```
-------------
|   |   |   |
-------------
|   | t | s |
-------------
| H |   | g |
-------------
Z - Special ability (Cooldown: 0 rounds)
L - Load save
S - Save game
Q - Quit game

Collision between occupant: turtle at (1,1) with attacker: sheep at (2,1)
Turtle reflects attack!
```

```
-------------
|   | s |   |
-------------
| t |   | g |
-------------
| H |   | g |
-------------
Z - Special ability (Cooldown: 0 rounds)
L - Load save
S - Save game
Q - Quit game

Collision between occupant: grass at (1,0) with attacker: sheep at (2,0)
Overtake!
Collision between occupant: sheep at (1,0) with attacker: turtle at (0,1)
Turtle reflects attack!
```

7. Antelope's Action() and Collision() function

Action() function of **Antelope** is the same as the default animal, but there is small change in **Move()** function, that instead of moving it by one cell, it moves antelope by **two** cells.

```
Position Antelope::Move(Commands dir, World* world)
{
    if (dir == NON) dir = (Commands)(rand() % 8);

    Position dest_position = position;
    switch (dir)
    {
    case WEST:
        if (dest_position.x <= 2) dest_position.x = 0;
        else dest_position.x-=2;
        break;
```

```
Position Animal::Move(Commands dir, World* world)
{
    if (dir == NON) dir = (Commands)(rand() % 8);

    Position dest_position = position;
    switch (dir)
    {
    case WEST:
        if (dest_position.x == 0) dest_position.x = 0;
        else dest_position.x--;
        break;
```

In **Collision()** function in case when **Antelope** is attacking stronger organism, or is attacked by stronger organism it has a chance to run away. (Only if there is free cell in neighborhood)

```
int rand_number = (rand() % 100);
if (rand_number >= CHANCE_TO_GET_AWAY)
{
    int occupant_cell = positionToCell(occupant->position, world);

    world->comment = world->comment + "Success!\n";
    world->cells[free_cell].isOccupied = true;
    world->cells[free_cell].occupant = occupant;
    occupant->position = world->cells[free_cell].position;

    int attacker_cell = positionToCell(attacker->position, world);
    world->cells[occupant_cell].occupant = attacker;
    attacker->position = world->cells[occupant_cell].position;

    world->cells[attacker_cell].isOccupied = false;
    world->cells[attacker_cell].occupant = NULL;
}
else
{
    world->comment = world->comment + "Failed!\n";
    Overtake(world, occupant, attacker);
}
```

```
-------------
|   |   | w |
-------------
| a |   |   |
-------------
|   | H |   |
-------------
Z - Special ability (Cooldown: 0 rounds)
L - Load save
S - Save game
Q - Quit game

Collision between occupant: antelope at (2,0) with attacker: wolf at (2,1)
There is a chance for Antelope to get away!
Success!
Collision between occupant: human at (1,2) with attacker: antelope at (1,0)
There is a chance for Antelope to get away!
Success!
```

Plants

```
class Plant : public Organism {
protected:
    int FindFreeNeighbor(World* world, const Organism* org);
    std::string speciesToString(Species sp);
    void Sow(World* world, const int cell, Species kind);
    void Overtake(World* world, Organism* occupant, Organism* attacker);
    char Draw() const override;
    void Collision(Organism* occupant, Organism* attacker, World* world) override;
public:
    Plant() {};
    void Action(World* world, Commands dir) override;
    ~Plant() {};
};
```

1. **Action() function**

```
void Plant::Action(World* world, Commands dir)
{
    this->allowed_to_move = false;

    const int free_cell = FindFreeNeighbor(world, this);
    if (free_cell != NOT_FOUND)
    {
        int rand_number = (rand() % 100);
        if (rand_number <= CHANCE_TO_SOW) Sow(world, free_cell, type);
    }
}
```

It's looking first for **free_cell**, if it's found there is a chance to **Sow**.

2. Collision() function

```
void Plant::Collision(Organism* occupant, Organism* attacker, World* world)
{
    world->comment = world->comment + "Collision between occupant: " + speciesToString(occupant->type)
        + " at " + occupant->position.toString() + " with attacker: " + speciesToString(attacker->type)
        + " at " + attacker->position.toString() + "\n";

    Overtake(world, occupant, attacker);
}
```

Default plant collision is to be **overtaken** by any organism that attacks it.

3. Sowing

```
void Plant::Sow(World* world, const int cell, Species kind)
{
    const int n = world->getN();
    const int m = world->getM();

    switch (this->type)
    {
    case GRASS:
        world->organisms.push_back(new Grass(n, m));
        break;
    case GUARANA:
        world->organisms.push_back(new Guarana(n, m));
        break;
    case SOW_THISTLE:
        world->organisms.push_back(new Sow_thistle(n, m));
        break;
    case BELLADONNA:
        world->organisms.push_back(new Belladonna(n, m));
        break;
    case SOSNOWKYS_HOGWEED:
        world->organisms.push_back(new Sosnowkys_hogweed(n, m));
        break;
    }

    const int last_index = world->organisms.size() - 1;
    world->organisms[last_index]->position = world->cells[cell].position;
    world->cells[cell].isOccupied = true;
    world->cells[cell].occupant = world->organisms[last_index];

    world->comment = world->comment + "New " + speciesToString(type) + " sowed at " +
        world->cells[cell].occupant->position.toString() + " from " + position.toString() + "\n";
}
```

It's really the same as the **Breed()** function in **animal**.

```
-------------
| g |   | H |
-------------
|   | g |   |
-------------
|   |   |   |
-------------
Z - Special ability (Cooldown: 0 rounds)
L - Load save
S - Save game
Q - Quit game

New grass sowed at (0,0) from (1,1)
```

4. **Sow thisle's Action() function**

```
void Sow_thistle::Action(World* world, Commands dir)
{
    this->allowed_to_move = false;
    int free_cell = FindFreeNeighbor(world, this);

    if (free_cell != NOT_FOUND)
    {
        int rand_number;
        bool result = false;

        for (int i = 0; i < SOW_ATTEMPTS; i++)
        {
            rand_number = (rand() % 100);
            if (rand_number <= CHANCE_TO_SOW)
            {
                result = true;
                break;
            }
        }
        if (result) Sow(world, free_cell, type);
    }
}
```

Firstly checks if there is **free_cell**, if so then it has 3 attempts to **Sow()**.

5. **Guarana's Collision() function**

```
void Guarana::Collision(Organism* occupant, Organism* attacker, World* world)
{
    world->comment = world->comment + "Collision between occupant: " + speciesToString(occupant->type)
        + " at " + occupant->position.toString() + " with attacker: " + speciesToString(attacker->type)
        + " at " + attacker->position.toString() + "\n";

    Overtake(world, occupant, attacker);
    attacker->strength += STRENGTH_INCREASE;

    world->comment = world->comment + "Guarana eaten, strength of animal: " + speciesToString(attacker->type)
        + " at " + attacker->position.toString() + " increased by 3!\n";

}
```

Collision() function is the same as default for **plant**, but it also increases the strength of attacker by 3.

```
-------------
| u |   |   |
-------------
|   | H |   |
-------------
|   |   |   |
-------------
Z - Special ability (Cooldown: 0 rounds)
L - Load save
S - Save game
Q - Quit game

Collision between occupant: guarana at (1,1) with attacker: human at (1,2)
Overtake!
Guarana eaten, strength of animal: human at (1,1) increased by 3!
```

6. Belladonna's Collision() function

```cpp
void Belladonna::Collision(Organism* occupant, Organism* attacker, World* world)
{
    world->comment = world->comment + "Collision between occupant: " + speciesToString(occupant->type)
        + " at " + occupant->position.toString() + " with attacker: " + speciesToString(attacker->type)
        + " at " + attacker->position.toString() + "\n";
    world->comment = world->comment + "Both organisms die!\n";

    if (attacker->type == HUMAN) world->player_alive = false;
    int attacker_cell = attacker->position.y * world->getM() + attacker->position.x;

    world->cells[attacker_cell].occupant = NULL;
    world->cells[attacker_cell].isOccupied = false;

    int occupant_cell = occupant->position.y * world->getM() + occupant->position.x;
    world->cells[occupant_cell].occupant = NULL;
    world->cells[occupant_cell].isOccupied = false;

    for (int i = 0; i < world->organisms.size(); i++)
    {
        if ((world->organisms[i]->position == occupant->position) && (world->organisms[i]->type == occupant->type))
        {
            delete world->organisms[i];
            world->organisms.erase(world->organisms.begin() + i);
            break;
        }
    }

    for (int i = 0; i < world->organisms.size(); i++)
    {
        if ((world->organisms[i]->position == attacker->position) && (world->organisms[i]->type == attacker->type))
        {
            delete world->organisms[i];
            world->organisms.erase(world->organisms.begin() + i);
            break;
        }
    }
}
```

If something attacks **belladonna**, both **belladonna** and attacker are **deleted**. Firstly I'm freeing the attacker's and occupant's cells, and then I'm looking for them in **world's organisms vector** and deleting them.

```
-------------
|   |   |   |
-------------
|   | b |   |
-------------
|   | H |   |
-------------
Game has been loaded from file!
```

```
-------------
|   |   |   |
-------------
|   |   |   |
-------------
|   |   |   |
-------------
Z - Special ability (Cooldown: 0 rounds)
L - Load save
S - Save game
Q - Quit game

Collision between occupant: belladonna at (1,1) with attacker: human at (1,2)
Both organisms die!
```

7. Sosnowky's hogweed Action() and Collision() function

```
void Sosnowkys_hogweed::Action(World* world, Commands dir)
{
    this->allowed_to_move = false;
    TerminateNeighbourhood(world);

    const int free_cell = FindFreeNeighbor(world, this);
    if (free_cell != NOT_FOUND)
    {
        int rand_number = (rand() % 100);
        if (rand_number <= CHANCE_TO_SOW) Sow(world, free_cell, type);
    }
}
```

Special behaviour of Sosnowky's hogweed is defined in TerminateNeighborhood() function.

```cpp
void Sosnowkys_hogweed::TerminateNeighbourhood(World* world)
{
    int destiny = 0;
    const Position pos = this->position;

    for (int i = pos.y - 1; i < pos.y + 2; i++)
    {
        if ((i < 0) || (i > world->getN() - 1)) continue;
        for (int j = pos.x - 1; j < pos.x + 2; j++)
        {
            int grid_number = i * world->getM() + j;
            if ((j < 0) || (j > world->getM() - 1)) continue;
            if (world->cells[grid_number].isOccupied && isAttackable(world->cells[grid_number].occupant->type))
            {
                if (world->cells[grid_number].occupant->type == HUMAN) world->player_alive = false;
                for (int i = 0; i < world->organisms.size(); i++)
                {
                    if ((world->organisms[i]->position == world->cells[grid_number].occupant->position)
                        && (world->organisms[i]->type == world->cells[grid_number].occupant->type))
                    {
                        world->comment = world->comment + "Animal " + speciesToString(world->organisms[i]->type)
                            + " has been terminated by hogweed at " + position.toString() + "\n";

                        delete world->organisms[i];
                        world->organisms.erase(world->organisms.begin() + i);
                        break;
                    }
                }
                world->cells[grid_number].occupant = NULL;
                world->cells[grid_number].isOccupied = false;
            }
        }
    }
}
```

I'm checking here the neighbors of **hogweed** and if it finds organism that is attackable (checked with **isAttackable()** function) it deletes it.

```
-------------
|   |   |   |
-------------
|   | y |   |
-------------
|   | H |   |
-------------
Game has been loaded from file!
```

```
-------------
|   |   |   |
-------------
|   | y |   |
-------------
|   |   |   |
-------------
Z - Special ability (Cooldown: 0 rounds)
L - Load save
S - Save game
Q - Quit game

Animal human has been terminated by hogweed at (1,1)
```

```cpp
void Sosnowkys_hogweed::Collision(Organism* occupant, Organism* attacker, World* world)
{
    world->comment = world->comment + "Collision between occupant: " + speciesToString(occupant->type)
        + " at " + occupant->position.toString() + " with attacker: " + speciesToString(attacker->type)
        + " at " + attacker->position.toString() + "\n";

    if (attacker->type == CYBER_SHEEP) Overtake(world, occupant, attacker);
    else
    {
        world->comment = world->comment + "Both organisms die!\n";
        if (attacker->type == HUMAN) world->player_alive = false;
        const int attacker_cell = attacker->position.y * world->getM() + attacker->position.x;

        world->cells[attacker_cell].occupant = NULL;
        world->cells[attacker_cell].isOccupied = false;

        const int occupant_cell = occupant->position.y * world->getM() + occupant->position.x;
        world->cells[occupant_cell].occupant = NULL;
        world->cells[occupant_cell].isOccupied = false;

        for (int i = 0; i < world->organisms.size(); i++)
        {
            if ((world->organisms[i]->position == occupant->position) && (world->organisms[i]->type == occupant->type))
            {
                delete world->organisms[i];
                world->organisms.erase(world->organisms.begin() + i);
                break;
            }
        }

        for (int i = 0; i < world->organisms.size(); i++)
        {
            if ((world->organisms[i]->position == attacker->position) && (world->organisms[i]->type == attacker->type))
            {
                delete world->organisms[i];
                world->organisms.erase(world->organisms.begin() + i);
                break;
            }
        }
    }
```

Collision() function is the same as **belladonna**, but it's checking if attacker == **Cyber Sheep**, if so then it's **overtaken**.

```
-------------
|   |   |   |
-------------
|   | y |   |
-------------
|   | H |   |
-------------
Game has been loaded from file!
```

```
-------------
|   |   |   |
-------------
|   |   |   |
-------------
|   |   |   |
-------------
Z - Special ability (Cooldown: 0 rounds)
L - Load save
S - Save game
Q - Quit game

Collision between occupant: sosnowkys hogweed at (1,1) with attacker: human at (1,2)
Both organisms die!
```

```
------------
| y |   |   |
------------
|   | c |   |
------------
|   |   |   |
------------
Z - Special ability (Cooldown: 0 rounds)
L - Load save
S - Save game
Q - Quit game

Collision between occupant: sosnowkys hogweed at (1,1) with attacker: cyber sheep at (2,1)
Overtake!
```

Human

1. **Initialize player**

```
player_alive = true;
special_ability_cnt = 0;
organisms.push_back(new Human(n, m));
const int human_pos = organisms[0]->position.y * m + organisms[0]->position.x;
cells[human_pos].isOccupied = true;
cells[human_pos].occupant = organisms[0];
```

It is initialized in **World constructor**. In every **Collision()** function that he might be killed there is line like this:

```
if (attacker->type == HUMAN) world->player_alive = false;
```

In **World's makeTurn()** function inside loop of performing actions:

```
int size = organisms.size();
for (int i = 0; i < size; i++)
{
    if (i >= organisms.size()) break;
    for (int j = 0; j < size; j++)
    {
        if (j >= organisms.size()) break;
        if (organisms[j]->allowed_to_move == true)
        {
            if (organisms[j]->type == HUMAN)
                organisms[j]->Action(this, dir);
            else
                organisms[j]->Action(this, NON);
            break;
        }
    }
}
```

dir is the key that we got earlier with **ConvertKey()** function. That's how movement of player is managed.

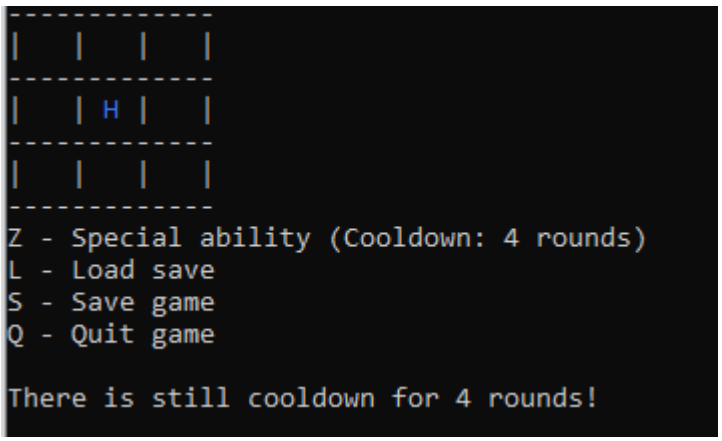
2. Special ability

```
void Human::Purification(World* world)
{
    int destiny = 0;
    const Position pos = this->position;

    for (int i = pos.y - 1; i < pos.y + 2; i++)
    {
        if ((i < 0) || (i > world->getN() - 1)) continue;
        for (int j = pos.x - 1; j < pos.x + 2; j++)
        {
            int grid_number = i * world->getM() + j;
            if ((j < 0) || (j > world->getM() - 1)) continue;
            if (world->cells[grid_number].isOccupied && world->cells[grid_number].occupant != this)
            {
                for (int i = 0; i < world->organisms.size(); i++)
                {
                    if ((world->organisms[i]->position == world->cells[grid_number].occupant->position)
                        && (world->organisms[i]->type == world->cells[grid_number].occupant->type))
                    {
                        world->comment = world->comment + "Organism " + speciesToString(world->organisms[i]->type)
                            + " has been terminated by human's purification at " + world->organisms[i]->position.toString() + "\n";

                        delete world->organisms[i];
                        world->organisms.erase(world->organisms.begin() + i);
                        break;
                    }
                }
                world->cells[grid_number].occupant = NULL;
                world->cells[grid_number].isOccupied = false;
            }
        }
    }
}
```

I've chosen to implement **purification**. It's pretty much the same as **Sosnowky's hogweed** action, but here it also destroys **plants**.

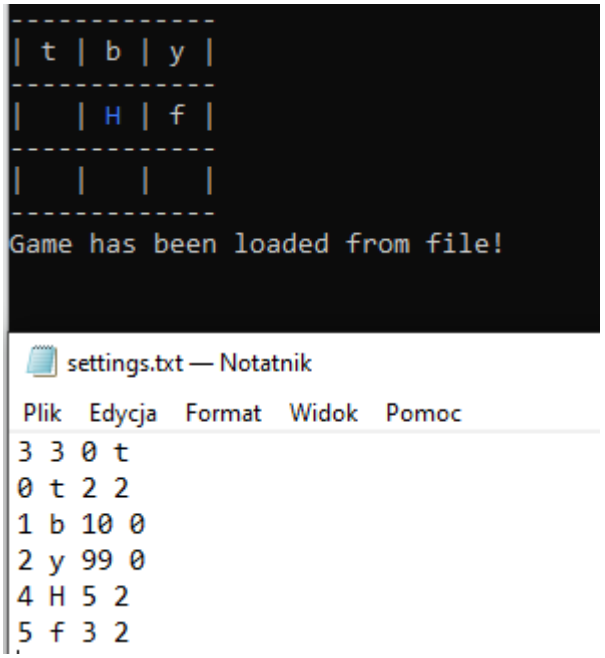
```
-------------
| t | b | y |
-------------
|   | H | f |
-------------
|   |   |   |
-------------
Game has been loaded from file!
```

```
-------------
|   |   |   |
-------------
|   | H |   |
-------------
|   |   |   |
-------------
Z - Special ability (Cooldown: 5 rounds)
L - Load save
S - Save game
Q - Quit game

Purification!
Organism turtle has been terminated by human's purification at (0,0)
Organism belladonna has been terminated by human's purification at (1,0)
Organism sosnowkys hogweed has been terminated by human's purification at (2,0)
Organism fox has been terminated by human's purification at (1,2)
```



Loading and saving

1. **File format**



First line in file are worlds settings : **m-n-special_ability_cooldown-palyer_alive**
Then in next lines are occupied cells in format: **cell index-type-strength-age**

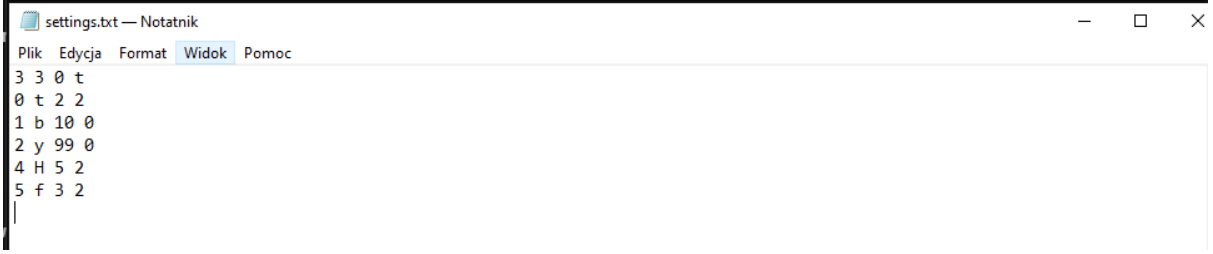
2. Loading

```
int World::loadGame()
{
    delete[] cells;
    cells = NULL;
    std::vector<Organism*>().swap(organisms);

    std::ifstream file("settings.txt");
    if (!file.good())
    {
        file.close();
        std::cout << "The settings file does not exist!" << std::endl;
        system("PAUSE");
        return -1;
    }

    int m, n, ab;
    char pa;
    file >> m >> n >> ab >> pa;
    if (!file.good())
    {
        file.close();
        std::cout << "Wrong format of file!" << std::endl;
        return -1;
    }

    this->m = m;
    this->n = n;

    cells = new Cell[n * m];
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            cells[(m * i) + j].position.x = j;
            cells[(m * i) + j].position.y = i;
            cells[(m * i) + j].occupant = NULL;
            cells[(m * i) + j].isOccupied = false;
        }
    }

    special_ability_cnt = ab;
    if (pa == 'f') player_alive = false;
    else if (pa == 't') player_alive = true;
    else
    {
        file.close();
        std::cout << "Wrong format of file!" << std::endl;
        return -1;
```

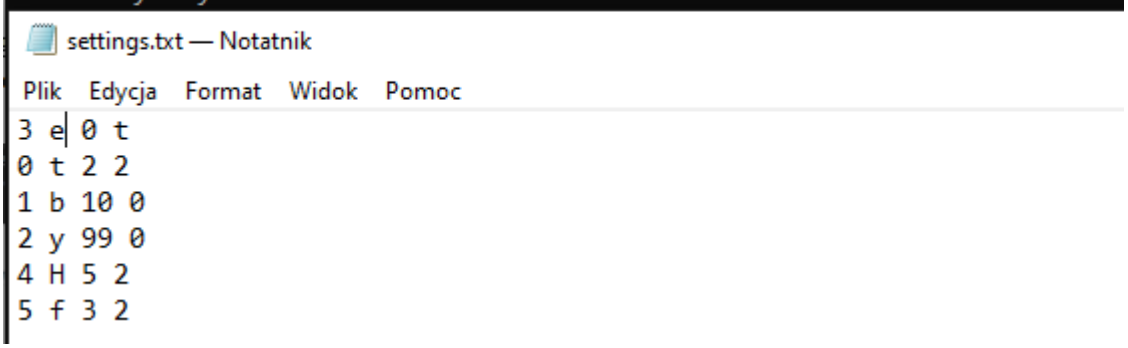
```
    int cell_index, strength, age;
    char type;
    while (file >> cell_index >> type >> strength >> age)
    {
        if (file.eof()) break;
        if (!file.good())
        {
            file.close();
            std::cout << "Wrong format of file!" << std::endl;
            return -1;
        }
        else
        {
            newOrganism(toSpecies(type), cell_index);
            const int last_index = organisms.size() - 1;
            organisms[last_index]->strength = strength;
            organisms[last_index]->age = age;
        }
    }
    file.close();
    return 0;
}
```

Firstly I'm **freeing** cells and organisms vector. Then I'm checking if the file even exists.
If so, I'm reading the first line and if everything went fine I'm creating **new cells map** with given **n and m,** and then initialize them with default values.
After this I'm reading every line till the end of file. If something went wrong during reading of file, the function **returns -1** which later proceeds in deleting the world and coming back to menu.

```
Game has been loaded succesfully!
Press any key to continue . . .
```




```
Wrong file format! Proceed to menu.
Press any key to continue . . .
```



3. Saving

```
void World::saveGame()
{
    std::ofstream file;
    file.open("settings.txt", std::ofstream::out | std::ofstream::trunc);

    file << m << " " << n << " " << special_ability_cnt << " ";
    if (player_alive == false) file << "f" << std::endl;
    else file << "t" << std::endl;

    for (int i = 0; i < n * m; i++)
    {
        if (cells[i].isOccupied)
        {
            file << i << " " << cells[i].occupant->Draw() << " " << cells[i].occupant->strength
                << " " << cells[i].occupant->age << std::endl;
        }
    }
    file.close();
}
```

Opening file and deleting its content. (If such file does not exist, it will create one)
Then printing world settings and later every occupied cell with information about occupant.

```
-------------
|   |   | t |
-------------
| h |   | s |
-------------
| H |   |   |
-------------
Game has been saved to file!
```

settings.txt — Notatnik

Plik Edycja Format Widok Pomoc

```
3 3 0 t
2 t 2 0
3 h 0 0
5 s 4 0
6 H 5 0
```

End of file
Thank you for your patience! :)