Федеральное государственное бюджетное образовательное
учреждение высшего образования
«Сибирский государственный университет телекоммуникаций и
информатики»
кафедра ПМиК

КУРСОВАЯ РАБОТА

по дисциплине «Объектно-ориентированное программирование»

Тема: « Жизненный цикл в заданной экосистеме »

Выполнил: студент группы ИС-341
Мильтов Данил Александрович

Проверил: преподаватель кафедры ПМиК
Бублей.Д.А

Новосибирск – 2024

Постановка задачи

Игроку предоставляется игровое поле размером 10 клеток на 10 клеток (реализованное в консоли). Каждый определенный период времени (например, 2 секунды) или по нажатию клавиши сменяется жизненный цикл. На клетках находятся живые организмы, определенных типов:

- Тип Растение: является пищей для травоядного животного; не двигается, появляется в незанятой клетке в случайном порядке раз в несколько жизненных циклов.

- Тип Травоядное животное: поглощает растения и является пищей для хищника; каждое травоядное двигается на случайную клетку поблизости раз в несколько жизненных циклов (период должен выбираться случайным образом), с некоторой вероятностью оставляя после себя травоядное животное. Наступая на клетку с растением поглощает его.

- Тип Хищник: поглощает травоядных, но не взаимодействует с растениями; каждый хищник двигается на случайную клетку поблизости раз в несколько жизненных циклов, поглощая всех травоядных, которых встретит; с растениями не взаимодействует (или взаимодействует, если это прописано в варианте задания, которые указаны ниже).

Способ размножения указывается в варианте. Травоядные и хищники обладают системой голода. Изначально у каждого из травоядных и хищников показатель голода равен определенному значению, указанному в вариантах ниже. Каждое перемещение живого организма отнимает 0,2 от показателя голода. Рождение нового живого организма отнимает 0,4. Поглощение другого вида прибавляет к голоду 0,2. В случае, когда состояние голода станет равно нулю – живое существо погибает. Растения живут ограниченное количество циклов, а затем погибают.

Технологии ООП:

1. Инкапсуляция

- Все поля данных (`x`, `y`, `hunger`, `lifeSpan` и т. д.) **приватны** или **защищены** (`protected`) в каждом классе.
- Внешние функции не имеют прямого доступа к полям данных объектов.

2. Наследование

- Базовый класс `Creature` является **абстрактным**, так как содержит только виртуальную функцию `moveAndAct`.
- Классы `Carrot`, `Rabbit` и `Wolf` наследуются от `Creature` и переопределяют метод `moveAndAct`.

3. Полиморфизм

- Используется **полиморфизм на основе виртуальных функций**:
  Метод `moveAndAct` вызывается через указатель или ссылку на базовый класс `Creature`. При этом конкретная реализация определяется типом объекта (например, `Rabbit` или `Wolf`).

4. Конструкторы

- У всех классов есть **конструкторы** для инициализации полей.
- Конструкторы используют **списки инициализации**, например:
  `Rabbit(int x, int y): Creature(x, y), hunger(1.0) {}`

5. Используемые технологии ООП

Статические элементы:

- **Статический генератор случайных чисел**:
  - Классы `Rabbit` и `Wolf` используют статический объект `std::mt19937 rng` для генерации случайных чисел.
  - Это позволяет разделить генератор между всеми экземплярами каждого класса.

Массивы указателей на объекты:

- Поле представляет собой двумерный массив (`std::vector<std::vector<std::shared_ptr<Creature>>>`), где хранятся указатели на объекты.

- Каждый элемент массива может быть объектом любого класса, унаследованного от `Creature`.

Использование объектов как возвращаемых значений:

- Методы, такие как `addNewCarrots`, работают с `std::shared_ptr<Creature>` и возвращают такие указатели, перемещая их по полю.

Приложение:

```
#include <iostream>
#include <vector>
#include <memory>
#include <cstdlib>
#include <ctime>
#include <random>
#include <thread>
#include <chrono>

constexpr int FIELD_SIZE = 10;
constexpr int MIN_WOLVES = 5;
constexpr int MIN_RABBITS = 5;

class Creature
{
protected:
    int x, y;

public:
    Creature(int x, int y) : x(x), y(y) {}
    virtual ~Creature() = default;
    virtual void
moveAndAct(std::vector<std::vector<std::shared_ptr<Creature>>> &field) = 0;
};

class Carrot : public Creature
{
private:
    int lifeSpan;
```

```
public:
    Carrot(int x, int y) : Creature(x, y), lifeSpan(5) {}

    void moveAndAct(std::vector<std::vector<std::shared_ptr<Creature>>>
&field) override
    {
        if (--lifeSpan <= 0)
        {
            field[x][y].reset();
        }
    }
};

class Rabbit : public Creature
{
private:
    float hunger;
    static std::mt19937 rng;

public:
    Rabbit(int x, int y) : Creature(x, y), hunger(1.0) {}

    void moveAndAct(std::vector<std::vector<std::shared_ptr<Creature>>>
&field) override
    {
        if (hunger <= 0)
        {
            field[x][y].reset();
            return;
        }

        std::uniform_int_distribution<int> dist(-1, 1);
        int dx = dist(rng);
        int dy = dist(rng);
        int nx = (x + dx + FIELD_SIZE) % FIELD_SIZE;
        int ny = (y + dy + FIELD_SIZE) % FIELD_SIZE;

        if (!field[nx][ny] || dynamic_cast<Carrot *>(field[nx][ny].get()))
        {
            if (dynamic_cast<Carrot *>(field[nx][ny].get()))
            {
                hunger += 0.2;
            }
            field[nx][ny] = field[x][y];
            field[x][y].reset();
```

```
            x = nx;
            y = ny;
        }

        hunger -= 0.2;
    }
};

std::mt19937 Rabbit::rng(std::random_device{}());

class Wolf : public Creature
{
private:
    float hunger;
    int eatenHares;
    bool hasReproduced;
    static std::mt19937 rng;

public:
    Wolf(int x, int y) : Creature(x, y), hunger(2.0), eatenHares(0),
hasReproduced(false) {}

    void moveAndAct(std::vector<std::vector<std::shared_ptr<Creature>>>
&field) override
    {
        if (hunger <= 0)
        {
            field[x][y].reset();
            return;
        }

        int moveRange = (hunger < 0.5) ? 2 : 1;
        std::uniform_int_distribution<int> dist(-moveRange, moveRange);

        for (int attempt = 0; attempt < 5; ++attempt)
        {
            int dx = dist(rng);
            int dy = dist(rng);

            if (dx == 0 && dy == 0)
                continue;

            int nx = (x + dx + FIELD_SIZE) % FIELD_SIZE;
            int ny = (y + dy + FIELD_SIZE) % FIELD_SIZE;

            if (!field[nx][ny] || dynamic_cast<Rabbit *>(field[nx][ny].get()))
```

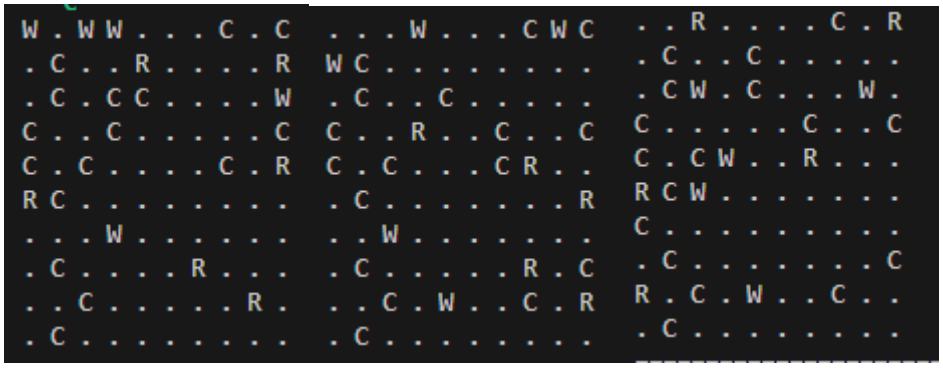
```
            {
                if (dynamic_cast<Rabbit *>(field[nx][ny].get()))
                {
                    hunger += 0.4;
                    ++eatenHares;
                    if (eatenHares > 2 && !hasReproduced)
                    {
                        hasReproduced = true;
                        field[x][y] = std::make_shared<Wolf>(x, y);
                    }
                }

                field[nx][ny] = field[x][y];
                field[x][y].reset();
                x = nx;
                y = ny;
                break;
            }
        }

        hunger -= 0.2;
    }
};

std::mt19937 Wolf::rng(std::random_device{}());

void addNewCarrots(std::vector<std::vector<std::shared_ptr<Creature>>> &field)
{
    for (int i = 0; i < 5; ++i)
    {
        int x = rand() % FIELD_SIZE;
        int y = rand() % FIELD_SIZE;
        if (!field[x][y])
        {
            field[x][y] = std::make_shared<Carrot>(x, y);
        }
    }
}

void ensureRabbitBalance(std::vector<std::vector<std::shared_ptr<Creature>>>
&field)
{
    int rabbitCount = 0;

    for (int i = 0; i < FIELD_SIZE; ++i)
    {
```

```
        for (int j = 0; j < FIELD_SIZE; ++j)
        {
            if (dynamic_cast<Rabbit *>(field[i][j].get()))
            {
                ++rabbitCount;
            }
        }
    }

    while (rabbitCount < MIN_RABBITS)
    {
        int x = rand() % FIELD_SIZE;
        int y = rand() % FIELD_SIZE;
        if (!field[x][y])
        {
            field[x][y] = std::make_shared<Rabbit>(x, y);
            ++rabbitCount;
        }
    }
}

void ensureMinimumWolves(std::vector<std::vector<std::shared_ptr<Creature>>>
&field)
{
    int wolfCount = 0;

    for (int i = 0; i < FIELD_SIZE; ++i)
    {
        for (int j = 0; j < FIELD_SIZE; ++j)
        {
            if (dynamic_cast<Wolf *>(field[i][j].get()))
            {
                ++wolfCount;
            }
        }
    }

    while (wolfCount < MIN_WOLVES)
    {
        for (int i = 0; i < FIELD_SIZE; ++i)
        {
            if (!field[i][0])
            {
                field[i][0] = std::make_shared<Wolf>(i, 0);
                ++wolfCount;
                break;
            }
```

```
            if (!field[i][FIELD_SIZE - 1])
            {
                field[i][FIELD_SIZE - 1] = std::make_shared<Wolf>(i,
FIELD_SIZE - 1);
                ++wolfCount;
                break;
            }
        }
    }
}

void printField(const std::vector<std::vector<std::shared_ptr<Creature>>>
&field)
{
    for (int i = 0; i < FIELD_SIZE; ++i)
    {
        for (int j = 0; j < FIELD_SIZE; ++j)
        {
            if (dynamic_cast<Carrot *>(field[i][j].get()))
                std::cout << "C ";
            else if (dynamic_cast<Rabbit *>(field[i][j].get()))
                std::cout << "R ";
            else if (dynamic_cast<Wolf *>(field[i][j].get()))
                std::cout << "W ";
            else
                std::cout << ". ";
        }
        std::cout << "\n";
    }
    std::cout << "========================\n";
}

int main()
{
    std::srand(std::time(0));
    std::vector<std::vector<std::shared_ptr<Creature>>> field(FIELD_SIZE,
std::vector<std::shared_ptr<Creature>>(FIELD_SIZE));

    for (int i = 0; i < 15; ++i)
    {
        int x = rand() % FIELD_SIZE;
        int y = rand() % FIELD_SIZE;
        field[x][y] = std::make_shared<Carrot>(x, y);
    }

    for (int i = 0; i < 10; ++i)
    {
```

```
        int x = rand() % FIELD_SIZE;
        int y = rand() % FIELD_SIZE;
        field[x][y] = std::make_shared<Rabbit>(x, y);
    }

    for (int i = 0; i < 5; ++i)
    {
        int x = rand() % FIELD_SIZE;
        int y = rand() % FIELD_SIZE;
        field[x][y] = std::make_shared<Wolf>(x, y);
    }

    while (true)
    {
        for (int i = 0; i < FIELD_SIZE; ++i)
        {
            for (int j = 0; j < FIELD_SIZE; ++j)
            {
                if (field[i][j])
                    field[i][j]->moveAndAct(field);
            }
        }

        addNewCarrots(field);
        ensureRabbitBalance(field);
        ensureMinimumWolves(field);

        printField(field);
        std::this_thread::sleep_for(std::chrono::seconds(2));
    }

    return 0;
}
```

Скриншоты:


W . W W . . . C . C
. C . . R . . . . R
. C . C C . . . . W
C . . C C
C . C . . . . C . R
R C . . . .
. . . W . . . . . .
. C . . . . R . . .
. . C . . . . . R .
. C . . . .
. . . W . . . C W C
W C . . . .
. C . . C . . . . .
C . . R . . C . . C
C . C . . . C R . .
. C . . . . R
. . W . . . .
. C . . . . . R . C
. . C . W . . C . R
. C . . . .
. . R . . . . C . R
. C . . C . . . . .
. C W . C . . . W .
C . . . . . C . . C
C . C W . . R . . .
R C W . . . .
C . . . .
. C C
R . C . W . . C . .
. C . . . .