Программирование на C++

Практическая работа №4. Консольные мини-игры (Часть 2)

Цели:

- Научиться проектировать логику игр, обеспечивающая чередование ходов, проверку победных условий и определение ничьей.
- Научиться использовать структуры и классы.

⚠️ Задания принимаются в формате ссылки на GitHub-репозиторий отправленной мне в dot с подписанием номера выполненной практической работы.

В репозитории должен находиться исходный код вашего задания, скриншоты или видео работы ваших мини-игр.

Пример оформления репозитория:

```
https://github.com/Meirbek-dev/SQLite_CourseProject
```

Мини-игра №3: «Крестики-нолики»

Задание

Разработать консольную игру **«Крестики-нолики».**

1. **Игровое поле:**
   - Представлено в виде матрицы 3×3.
   - Изначально поле заполняется номерами ячеек (1-9) или символами-пробелами, чтобы игроки понимали позиции для ввода.
2. **Игроки:**
   - Играют два игрока, один играет крестиками (`x`), а второй — ноликами (`o`).
   - Игроки по очереди выбирают свободную клетку, вводя её номер.
3. **Основной игровой процесс:**
   - После каждого хода программа очищает (или перерисовывает) игровое поле и отображает его текущее состояние.
   - Программа проверяет, не образовалась ли выигрышная комбинация (три одинаковых символа в строке, столбце или по диагонали) либо ничья.
   - Если игрок выиграл, выводится сообщение с именем или символом победителя.
   - В случае ничьей (нет свободных ячеек, и ни у кого нет выигрышной комбинации) программа выводит соответствующее сообщение.
4. **Повтор игры:**
   - После завершения игры пользователю предлагается возможность начать новую игру.
5. **Обработка ввода:**
   - Программа должна проверять корректность ввода:
     - Если введён номер, соответствующий уже занятой ячейке или неподходящий номер (не в диапазоне от 1 до 9), игрок должен увидеть сообщение об ошибке и быть предложено ввести новое значение.

- Обработка нечислового ввода (например, ввод символов вместо числа).

Этапы выполнения

1. Подключение необходимых заголовочных файлов

Используем стандартные библиотеки для ввода-вывода, работы со строками и для очистки экрана.

```
#include <iostream>
#include <vector>
#include <string>
```

---

2. Объявление структуры данных для игрового поля

- Идея: использовать массив или вектор строк размером 3x3 для хранения символов.
- Можно организовать игровое поле как одномерный вектор из 9 элементов, где позиции соответствуют номерам от 1 до 9.

---

3. Функция для отображения игрового поля

Создайте функцию `printField()`, которая выводит текущее состояние поля в консоли.

```
void printField(const std::vector<char>& field) {
    // TODO
}
```

---

4. Функция для проверки победы

Реализуем проверку выигрышных комбинаций. Создадим функцию `checkWin()`, которая возвращает `true`, если найдена победная позиция.

```
bool checkWin(const std::vector<char>& field, char player) {
    // Выигрышные комбинации:
    const int winCombos[8][3] = {
        {0, 1, 2}, // первая строка
        {3, 4, 5}, // вторая строка
        {6, 7, 8}, // третья строка
        {0, 3, 6}, // первый столбец
        {1, 4, 7}, // второй столбец
        {2, 5, 8}, // третий столбец
        {0, 4, 8}, // главная диагональ
        {2, 4, 6}  // побочная диагональ
    };

    // TODO
}
```

---

5. Основной игровой цикл

Реализуем основной цикл игры в функции `main`. Здесь будет:

- Переменная для хранения текущего игрока (`X` или `O`).
- Переменная для отслеживания количества ходов.
- Обработка ввода игрока:
  - Проверка, что введённое значение является числом.
  - Проверка, что число попадает в диапазон от 1 до 9 и выбранная клетка ещё не занята.
- После каждого хода происходит:
  - Проверка на победу.
  - Если поле заполнено (9 ходов) без победителя — объявляется ничья.

---

6. Дополнительные рекомендации и улучшения

1. **Очистка экрана:**

- Для улучшения визуального восприятия можно добавить функцию очистки экрана перед отображением поля.
- Например, на системах Unix/Linux можно использовать `system("clear")`, а на Windows — `system("cls")` .

> Примечание: А можно просто выводить несколько пустых строк.

2. **Улучшение обработки ввода:**
   - Можно написать отдельную функцию для получения и проверки ввода от пользователя, чтобы избежать дублирования кода.
3. **Дополнительные опции:**
   - Добавить режим игры: человек против компьютера со случайными ходами.

Примечания к заданию

1. **Использование вектора для игрового поля:**
   - Вектор `field` инициализируется номерами от 1 до 9, что позволяет игроку видеть, какие клетки доступны для хода.
2. **Функция отображения игрового поля `printField`:**
   - Функция выводит поле в привычном формате 3×3, что помогает игрокам видеть текущее состояние игры.
3. **Проверка выигрышных комбинаций:**
   - Функция `checkWin` перебирает все возможные выигрышные линии (строки, столбцы, диагонали) и сравнивает их с символом текущего игрока.
4. **Основной игровой цикл в функции `main`:**
   - Цикл игры продолжается до тех пор, пока не завершится текущая игра (выигрыш или ничья).

- После каждого хода происходит проверка на корректность ввода, занятость клетки и затем переход хода.
- После завершения игры пользователю предлагается сыграть ещё раз.

5. **Обработка ошибок ввода:**
   - Используйте проверу `std::cin.fail()` для обнаружения неверного ввода (например, если введены символы вместо числа).
   - В случае ошибки состояние потока сбрасывается, а ввод очищается.

Вопросы для самостоятельного обсуждения

1. Как ваша программа обрабатывает некорректный ввод, например, если пользователь вводит символ вместо числа или число вне диапазона от 1 до 9?
2. Как можно реализовать возможность игры человека против компьютера или компьютера против компьютера?
3. Как можно модифицировать программу для ведения статистики игр (например, количество побед каждого игрока, ничьи)?
4. Как реализована функция `printField()` и как она обеспечивает корректное отображение игрового состояния?
5. Как работает функция `checkWin()`? Опишите, каким образом проверяются все выигрышные комбинации.
6. Какие улучшения можно внести для работы игры в случае некорректного ввода символов вместо чисел?
7. Как реализована логика смены ходов между игроками (`X` и `O`)?
8. Как программа определяет, что игра закончилась вничью? Какие условия для этого существуют?
9. Если бы вы захотели изменить размер игрового поля (например, на 4×4), какие изменения потребовались бы в коде?

Мини-игра №4: «Битва с монстром»

Задание

Разработать консольную игру **«Битва с монстром»**, в которой:

1. **Персонажи:**
    - Создать классы или структуры для описания главного героя и монстра.
    - Каждый персонаж должен иметь характеристики, например, здоровье (health), силу атаки (attack power), защиту (defense) и, при желании, специальные умения.
2. **Бой:**
    - Игра представляет собой пошаговое сражение между героем и монстром.
    - В каждом ходе игрок может выбрать атаку, которая наносит случайное количество урона и выбрать атаковать или использовать заклинание.
    - Монстр отвечает атакой согласно своей логике (может быть реализован случайным образом).
3. **Отображение состояния боя:**
    - После каждого хода программа выводит текущие показатели здоровья обоих участников.
    - При необходимости можно отобразить дополнительные данные (например, количество оставшихся заклинаний или бонусные очки).
4. **Завершение игры:**
    - Игра завершается, когда здоровье героя или монстра становится меньше или равно нулю.
    - После окончания игры выводится сообщение о победе (если побеждает герой) или поражении (если побеждает монстр).
5. **Повтор игры:**
    - После завершения сражения программа предлагает пользователю возможность начать новую битву.

6. **Обработка ввода:**
    - Программа должна проверять корректность ввода (например, если пользователь выбирает действие, то ввод должен соответствовать доступным номерам/командам).
    - В случае неверного ввода пользователю предлагается повторить выбор.

Этапы выполнения

1. Подключение необходимых заголовочных файлов

Используем стандартные библиотеки для ввода/вывода, работы со строками, векторами и генерации случайных чисел.

---

2. Определение классов/структур для персонажей

1. **Структура (или класс)** `Character`**:**
    - Поля: имя (`std::string name`), здоровье (`int health`), сила атаки (`int attack`) и.т.д..
    - Метод: функция для нанесения удара (например, `attackTarget(Character &target)`).

Пример:

```
struct Character {
    std::string name;
    int health;
    int attack;
    int defence;

    // Функция атаки: наносит урон цели
    void attackTarget(Character &target) {
        std::cout << name << " атакует " << target.name << " и наносит "
                  << attack << " единиц урона.\\n";
        target.health -= attack;
```

```
    }
};
```

3. Функция для отображения состояния боя

Создайте функцию `printStatus()` для вывода текущего состояния героя и монстра.

```
void printStatus(const Character &hero, const Character &monster) {
    std::cout << "\\n--- Статус сражения ---\\n";
    std::cout << hero.name << " (Здоровье: " << hero.health << "\n3ащита: "
<< hero.defence <<  ")\\n";
    std::cout << monster.name << " (Здоровье: " << monster.health << "\n3ащ
ита: " << monster.defence << ")\\n";
    std::cout << "------------------------\\n\\n";
}
```

4. Основной игровой цикл

Реализуйте основной цикл сражения в функции `main()`. Здесь будет:

- Инициализация персонажей с помощью объектов типа `Character`.
- Организация цикла, который продолжается до тех пор, пока здоровье героя или монстра больше нуля.
- Обработка ввода пользователя:
    - Предоставьте выбор действий (например, «1. Атаковать», «2. Пропустить ход»).
    - Проверка корректности выбора.
- В цикле:
    - Герой совершает действие.
    - Если монстр всё ещё жив, он отвечает атакой (например, автоматически или случайно, если реализовано более одной стратегии).

- Вывод текущего состояния с помощью функции `printStatus()`.

---

5. Дополнительные задания

1. **Очистка экрана:**
   - Добавьте функцию `clearScreen()`, которая очищает консоль перед обновлением статуса боя (например, с помощью `system("cls")` для Windows или `system("clear")` для Unix/Linux).
2. **Расширение функционала:**
   - Добавьте дополнительные типы атак, специальные умения или возможность использования предметов.
3. **Обработка ввода:**
   - Вынесите проверку пользовательского ввода в отдельную функцию для избежания дублирования кода.
4. **Структурирование кода:**
   - Организуйте код в виде структур/классов и функций, чтобы упростить дальнейшее расширение игры.
5. **Логирование и статистика:**
   - Реализуйте возможность ведения статистики (количество сыгранных битв, побед героя и проигрышей).

Примечания к заданию

1. **Использование классов/структур для персонажей:**
   - Использование структуры `Character` облегчает хранение и управление данными о каждом персонаже (имя, здоровье, атака).
   - Метод `attackTarget()` осуществляет атаку, уменьшая здоровье цели, что иллюстрирует принцип инкапсуляции.

Вопросы для самостоятельного обсуждения

1. Какие преимущества дает использование классов/структур для организации кода игры?
2. Как можно расширить игру, добавив дополнительные типы атак или специальные умения героя и монстра?
3. Каким образом можно реализовать режим игры, где монстр обладает разными стратегиями атаки?
4. Как можно улучшить отображение статуса боя (например, добавить ASCII-графику или цветной текст)?
5. Какие подходы существуют для ведения статистики сыгранных боев и как их можно реализовать в этом проекте?
6. Как вы имплементировали защиту персонажей?
7. Что произойдет, если здоровье персонажа станет отрицательным, и как этого можно избежать?