Семинар #3: Динамический полиморфизм. Домашнее задание.

Задача 1. Вызов методов

Пусть есть следующие классы `Alice` и `Bob`. Метод `speak` класса `Bob` скрывает метод `speak` класса `Alice`:

```
struct Alice
{
    void speak() {std::cout << "Alice" << std::endl;}
};

struct Bob : public Alice
{
    void speak() {std::cout << "Bob" << std::endl;}
};
```

Что будет напечатано (или произойдёт ошибка), если эти классы будут использованы следующим образом:

1. ```
   Alice a;
   Bob b = a;
   b.speak();
   ```

2. ```
   Bob b;
   Alice a = b;
   a.speak();
   ```

3. ```
   Alice a;
   Bob* pb = &a;
   pb->speak();
   ```

4. ```
   Bob b;
   Alice* pa = &b;
   pa->speak();
   ```

Пусть есть следующие классы `Alice` и `Bob`. Метод `speak` класса `Bob` переопределяет метод `speak` класса `Alice`:

```
struct Alice
{
    virtual void speak() {std::cout << "Alice" << std::endl;}
};

struct Bob : public Alice
{
    void speak() {std::cout << "Bob" << std::endl;}
};
```

Что будет напечатано (или произойдёт ошибка), если эти классы будут использованы следующим образом:

5. ```
   Alice a;
   Bob b = a;
   b.speak();
   ```

6. ```
   Bob b;
   Alice a = b;
   a.speak();
   ```

7. ```
   Alice a;
   Bob* pb = &a;
   pb->speak();
   ```

8. ```
   Bob b;
   Alice* pa = &b;
   pa->speak();
   ```

9. ```
   Alice* pa = new Bob;
   pa->speak();
   delete pa;
   ```

10. ```
    std::unique_ptr<Alice> pa = std::make_unique<Bob>();
    pa->speak();
    ```

Пусть есть следующие классы `Alice`, `Bob` и `Casper`:

```
struct Alice
{
    virtual void speak() {std::cout << "Alice" << std::endl;}
};

struct Bob : public Alice
{
    void speak() {std::cout << "Bob" << std::endl;}
};

struct Casper : public Bob
{
    void speak() {std::cout << "Casper" << std::endl;}
};
```

Что будет напечатано (или произойдёт ошибка), если эти классы будут использованы следующим образом:

11. ```
    Casper c;
    Alice* pa = &c;
    pa->speak();
    ```

12. ```
    Casper c;
    Bob* pb = &c;
    pb->speak();
    ```

Пусть есть следующие классы `Alice`, `Bob` и `Casper`:

```
struct Alice
{
    virtual void speak() {std::cout << "Alice" << std::endl;}
};

struct Bob
{
    void speak() {std::cout << "Bob" << std::endl;}
};

struct Casper : public Alice, public Bob
{
    void speak() {std::cout << "Casper" << std::endl;}
};
```

Что будет напечатано (или произойдёт ошибка), если эти классы будут использованы следующим образом:

13. ```
    Casper c;
    Alice* pa = &c;
    pa->speak();
    ```

14. ```
    Casper c;
    Bob* pb = &c;
    pb->speak();
    ```

Для того, чтобы сдать эту задачу нужно создать файл в формате `.txt` и, используя любой текстовый редактор, записать в него ответы в следующем формате (ответы ниже неверны):

```
1) Alice
2) Bob
3) Error
```

После этого, файл нужно поместить в ваш репозиторий на github.

Задача 2. Личности

В `individuals.cpp` написан интерфейс `Individual` и классы `Alice`, `Bob` и `Casper`, реализующие этот интерфейс.

Подзадачи

1. **Вектор личностей**
   Напишите функцию `createIndividuals`, которая будет создавать вектор из различных личностей и возвращать его. Более точно, вектор должен иметь тип `std::vector<Individual*>`. Функция должна создавать в куче 3 объекта типа `Alice`, 2 объекта типа `Bob` и 4 объекта типа `Casper` и помещать указатели на эти объекты в вектор. После этого функция должна возвращать этот вектор.

2. **Вызов виртуальных методов**
   Напишите функцию `letThemSpeak`, которая принимает на вход вектор типа `std::vector<Individual*>` (по ссылке) и вызывает метод `speak` у каждого объекта в векторе. Протестируйте функции `createIndividuals` и `letThemSpeak`.

3. **Удаление личностей**
   Напишите функцию `deleteIndividuals`, которая принимает на вход вектор типа `std::vector<Individual*>` и освобождает память, выделенную функцией `createIndividuals`. Нужно использовать оператор `delete` на каждый указатель.

4. **Идентификация**
   Напишите функцию `int identification(Individual* p)`, которая должна возвращать:

   - 0 – если указатель `p` указывает на объект типа `Alice`.
   - 1 – если указатель `p` указывает на объект типа `Bob`.
   - 2 – если указатель `p` указывает на объект типа `Casper`.

   Решите эту подзадачу тремя способами:

   (a) Дописав ещё один виртуальный метод во все классы. Этот метод должен возвращать нужное число.

   (b) Не изменяя классы. Используя оператор приведения `dynamic_cast`.

   (c) Не изменяя классы. Используя оператор `typeid`.

5. **Личности и умные указатели**
   Напишите новые версии функций `createIndividuals` и `letThemSpeak`, которые будут использовать умные указатели `std::unique_ptr<Individual>` вместо обычных указателей `Individual*`. Сделайте эту подзадачу в отдельном файле.

Задача 3. Виджеты

В папке `widgets` содержится программа, написанная с использованием библиотеки SFML, которая помещает в окне несколько элементов интерфейса трёх видов: кнопки (класс `Button`), слайдеры (класс `Slider`) и перетаскиваемые таблички (класс `Draggable`). Однако, на данный момент классы виджетов никак друг с другом не связаны. Поэтому, для работы с ними пришлось создать 3 разных вектора (`buttons`, `sliders` и `draggables`), что не совсем удобно. Более того, если бы в программе добавился ещё один виджет, то придётся добавлять ещё один вектор и дописывать дополнительные циклы во многих местах в программе.

Для того, чтобы решить описанные выше проблемы, объедините классы `Button`, `Slider` и `Draggable` в единую иерархию наследования. Напишите новый интерфейс `Widget`. Все классы виджетов должны наследоваться от этого интерфейса. После этого объедините все вектора в один вектор, содержащий элементы типа *указатель на виджет*. Решите эту задачу в двух вариантах:

1. С использованием обычных указателей.

2. С использованием умных указателей (`std::unique_ptr` или `std::shared_ptr` на ваш выбор).

Задача 4. Дерево навыков

В папке `skilltree` находится реализация простого дерева навыков для компьютерной игры.

- `Node` – абстрактный класс, который описывает узел дерева навыков. Узел может быть в трёх состояниях
  - `Blocked` – заблокированный узел. При нажатии на такой узел ничего не должно происходить.
  - `Unblocked` – разблокированный узел. При нажатии на такой узел он активируется. При активации узла, все его дети разблокируются.
  - `Activated` – активированный узел.
- `HitNode` – абстрактный класс, наследник класса `Node`, который описывает круглый переключаемый узел дерева навыков. При нажатии на этот узел, если он разблокирован, узел активируется. При повторном нажатии на этот узел, он деактивируется и все его потомки становятся заблокированными.
- `ShieldSkillNode`, `HandSkillNode` и другие – конкретные классы переключаемых узлов.

Подзадачи

1. **Новый навык** – добавьте новый навык – огненный шар (файл изображения `icon_fireball.png`). Добавьте этот навык в дерево навыков.
2. **Накопительный навык** – создайте новый узел `AccumulativeNode`, наследник класса `Node`. В отличии от узла `HitNode` в этот узел можно вкладывать несколько очков навыка. При нажатии на этот навык, если он разблокирован, он становится активированным, а все его дети разблокируются. При повторном нажатии на этот левой кнопкой мыши у этого узла увеличивается уровень активации до некоторого максимального значения. При нажатии правой кнопкой мыши уровень ативации должен уменьшаться на 1. Если уровень достигнет нуля, то узел перестаёт быть активированным, а все его потомки становятся заблокированными.

   

   Визуально этот узел должен быть квадратным (учтите это при определении столкновения при нажатии кнопки мыши). Внизу этого узла должен быть написан уровень активации в виде `a / b`, где `a` – текущий уровень активации, а `b` – максимальный уровень активации.
3. **Наследники накопительного навыка** – создайте 2 класса наследника `AccumulativeNode`. Изображения для этих навыков можно найти в папке `icons` (`icon_rect_sword.png` и `icon_rect_chain.png`). Добавьте эти узлы в дерево навыков.
4. **Абстрактное дерево навыков** – создайте абстрактный класс дерева навыков `SkillTree`. Этот класс должен описывать всё дерево навыков, а не один узел. Соответственно, объекты этого класса должны содержать указатель на корень дерева. Также они должны хранить количество свободных очков навыка. При активации какого-либо навыка количество свободных очков навыка должно уменьшаться на 1. При деактивации навыка, все свободные очки должны возвращаться. Если свободных очков нет, то получить новый навык нельзя. Количество свободных очков навыка должно также отображаться на экране.
5. **Конкретное дерево навыков** – создайте класс `MageSkillTree`, наследник абстрактного класса `SkillTree`. Этот класс должен создавать дерево навыков для персонажа-мага. Дерево должно содержать как узлы типа `HitNode` так и узлы типа `AccumulativeNode`.

При решении этой задачи используйте умные указатель `std::shared_ptr`.