

## Лабораторная работа №2

### Простые классы

#### Контрольные вопросы:

1. Что такое объектно-ориентированное программирование? Какова цель внедрения ООП?
2. Что такое объект и что такое класс?
3. Дайте определение принципа ООП абстракции и инкапсуляции.
4. Дайте определение принципа ООП наследования и полиморфизма.
5. Какие достоинства и недостатки ООП вы знаете?
6. Дайте краткие сведения о языке C++. Чем он отличается от C?
7. Для чего нужны пространства имен и как их объявлять?
8. Что такое потоки ввода-вывода и как с ними работать в C++?
9. Для чего нужны using-объявления?
10. Какие уровни доступа есть при объявлении класса? Формат объявления класса?
11. Что такое конструктор и деструктор? Формат объявления конструктора?
12. Что такое указатель this и в каких случаях он применяется?
13. Зачем нужна интерфейсная часть и часть реализации при объявлении класса в программе?
14. Какие рекомендации существуют по объявлению классов?
15. Что такое юнит-тестирование? На какие части обычно разделен код теста?

#### Задание

В данной лабораторной работе необходимо преобразовать лабораторную работу №1 таким образом, чтобы заданная сущность была представлена в виде класса. Класс должен быть обязательно оформлен отдельными файлами описания (.h) и реализации (.cpp). Данный класс должен иметь конструктор с параметрами, необходимыми для создания сущности, методы для реализации всех нужных операций над сущностью. Доступ к полям класса должен быть приватным, извне доступ к ним осуществляется исключительно через геттеры и сеттеры класса. Например, для структуры Person из предыдущей лабораторной это будет выглядеть следующим образом:

```
class Person
{
    char name[30];
    int yearBirth;
    int age; // поле, которое будет рассчитано при создании

public:
    Person(const char* name, int yearBirth)
    {
        setName(name);
        this->yearBirth = yearBirth;
        age = getCurrentYear() - yearBirth; //рассчитываем поле age
    }

    //age и yearBirth нельзя менять после создания объекта,
    //поэтому они имеют только геттеры
    int getAge() { return age; }
    int getYearBirth() { return yearBirth; }

    //человек может сменить имя, поэтому есть и геттер и сеттер
    const char* getName() { return name; }
    void setName(const char* name) { strcpy(this->name, name); }
};
```

Также в данном задании необходимо написать класс юнит-тестов для каждого нетривиального метода созданного класса (см. ниже «Юнит-тестирование»). **Количество тестов должно быть не менее 10.** Результаты работы юнит-тестов необходимо выводить в консоль. Функция `main` должна содержать только запуск всех юнит-тестов.

Каждый тест должен быть написан по принципу **arrange -> act -> assert** и осуществлять вывод в консоль следующей информации:

- Название теста (какая функция тестируется)
- Исходные данные
- Ожидаемый результат операции
- Фактический результат операции
- Тест пройден \ Тест провален

В конце класс юнит-тестов должен вывести количество пройденных и проваленных тестов.

## Юнит-тестирование

Модульное тестирование или юнит-тестирование (unit testing) — процесс позволяющий проверить на корректность отдельные модули исходного кода программы вместе с соответствующими управляющими данными, процедурами использования и обработки.

Идея юнит-тестирования состоит в том, чтобы писать тесты для каждой **нетривиальной** функции или метода. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к регрессии, то есть к появлению ошибок в уже оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок. Например, обновить используемую в проекте библиотеку до актуальной версии можно в любой момент, прогнав тесты и выявив несовместимости.

Юнит-тестирование – это первый рубеж на борьбе с ошибками в коде программ. За ним следует еще интеграционное, приемочное и ручное тестирование (в том числе «свободный поиск»). Любой долгосрочный проект без надлежащего покрытия тестами обречен рано или поздно быть переписанным с нуля.

Каждый юнит-тест должен быть атомарным, т.е. один тест проверяет только одну функцию. Юнит-тест является спецификацией метода класса, контрактом: какие входные параметры ожидает этот метод, и что остальные компоненты системы ждут от него на выходе.

Написание любого юнит-теста начинается с выбора его имени. Один из рекомендуемых подходов – формировать его имя из трех частей:

- имя тестируемой рабочей единицы
- сценарий теста
- ожидаемый результат

Таким образом название читается как завершенное предложение, а это повышает простоту работы с тестами. Чтобы понять, что тестируется, достаточно, без вникания в логику работы кода, просто прочитать названия тестов.

Но в ряде случаев с логикой работы тестового кода все-таки нужно разбираться. Чтобы упростить эту работу, структуру юнит-теста можно формировать из трех частей:

- **Arrange** — здесь производится создание и инициализация требуемых для проведения теста объектов
- **Act** — собственно проведение тестируемого действия
- **Assert** — здесь производится сравнение полученного результата с эталонным

Для того чтобы повысить читабельность тестов рекомендуется эти части отделять друг от друга пустой строкой. Это ориентирует тех, кто читает ваш код, и поможет быстрее найти ту часть теста, которая их интересует больше всего.

Приведём пример класса для юнит-тестирования: **см. проект `UnitTests.dev`**