[Лабораторная работа №3. Проектирование собственных классов 2](#_Toc463983136)

[Использование объектно-ориентированного подхода при разработке программ. Разработка программ через тестирование. 2](#_Toc463983137)

[Анализ предметной области решаемой задачи 2](#_Toc463983138)

[Использование классов для моделирования сущностей предметной области 2](#_Toc463983139)

[Проектирование интерфейсной части класса 3](#_Toc463983140)

[Выбор необходимого уровня абстракции 3](#_Toc463983141)

[Вспомогательные средства Microsoft Visual Studio для добавления класса 3](#_Toc463983142)

[Запись интерфейсной части класса 5](#_Toc463983143)

[Разработка через тестирование 5](#_Toc463983144)

[Резюме 9](#_Toc463983145)

[Практические задания 10](#_Toc463983146)

[Обязательные задания 10](#_Toc463983147)

[Задание 1 10](#_Toc463983148)

[Вариант 1 – Телевизор – 50 баллов 10](#_Toc463983149)

[Вариант 2 – Автомобиль – 140 баллов 11](#_Toc463983150)

[Вариант 3 – Прямоугольник - 50 баллов 12](#_Toc463983151)

[Дополнительные задания 16](#_Toc463983152)

[Задание 2 – Калькулятор – 300 баллов 16](#_Toc463983153)

[Формат входных и выходных данных 16](#_Toc463983154)

[Примеры 17](#_Toc463983155)

[Бонус в 200 баллов за оптимизацию вычислений функций 19](#_Toc463983156)

[Бонус в 100 баллов за возможность вычисления очень больших последовательностей функций 19](#_Toc463983157)

[Ссылки 19](#_Toc463983158)

# Лабораторная работа №3. Проектирование собственных классов

## Использование объектно-ориентированного подхода при разработке программ. Разработка программ через тестирование.

### Анализ предметной области решаемой задачи

Рассмотрим задачу: имеется стакан определенной емкости. Имеется определенное количество жидкости заданного объема. В стакан можно налить некоторое количество жидкости, а также вылить из него некоторое количество жидкости. Очевидно, что в стакан нельзя поместить жидкости больше, чем в нем есть места, также из стакана нельзя вылить жидкости больше, чем имеется в наличии.

Разработать программу, позволяющую пользователю вводить заданные команды с клавиатуры, при помощи которых пользователь может наливать в стакан жидкость и выливать ее из стакана, а также узнать количество жидкости, фактически находящейся в стакане и фактически вылитое из него.

Для решения данной задачи можно было бы воспользоваться обычными целочисленными переменными, хранящими емкость стакана и объем находящейся в нем воды, а также разработать несколько функций для выполнения основных операций над ними.

Одним из очевидных недостатков такого решения программы будет следующий - потребуется кардинальная переделка программы в случае, если задача будет иметь дело не с одним стаканом, а с произвольным их количеством. В этом случае ранее написанный код окажется попросту бесполезным.

Улучшить ситуацию можно было бы, объявив структуру **Glass**, объединяющую данные о емкости стакана и объеме содержащейся в нем жидкости, а также функции, осуществляющие операции над данной структурой.

struct Glass

{

int capacity;

int amountOfWater;

};

void CreateEmptyGlass(Glass & glass, int capacity);

int AddWater(Glass & glass, int amount);

int PourWater(Glass & glass, int amount);

Тем не менее, у пользователя данных классов по-прежнему сохраняется прямой доступ к полям структуры Glass в обход имеющихся функций, что позволяет умышленно или случайно задать им недопустимые значения, например, отрицательную вместимость или количество воды, превышающее вместимость стакана.

Еще одна проблема, связанная с возможностью прямого доступа к полям структур, - это привязка внешнего кода к деталям ее реализации, что существенно ограничивает свободу по их расширению. В нашем случае наличие поля **amountOfWater** в структуре Glass выйдет нам боком, если в дальнейшем понадобиться «наливать» в стакан помимо воды еще и масло. В этом случае нам придется переписать весь клиентский код, который для определения объема жидкости в стакане использует значение поля **amountOfWater**, т.к. в случае, когда стакан содержит и масло и воду, объем жидкости в нем будет складываться из объема воды и масла.

### Использование классов для моделирования сущностей предметной области

Классы – как раз тот механизм объектно-ориентированных языков программирования, позволяющий не только представить сущности предметной области на нужном уровне абстракции, но и ограничить набор действий которые можно совершать над ними, а также скрыть детали их внутренней реализации от доступа извне.

Объявление класса в языке Си++ во многом схоже с объявлением структуры. Класс объявляется при помощи ключевого слова **class**. В нашем случае объявление класса CGlass может быть сделано следующим образом:

class CGlass

{

// поля класса

};

Поля класса – это данные и методы, которыми будут обладать все **объекты**, или как их еще называют – **экземпляры**, данного класса.

Проектирование класса следует начинать с его интерфейсной части, т.е. того, как будет представлен класс внешнему миру, нежели с его внутреннего устройства. Иными словами, интерфейс класса должен определять внутреннее устройство класса, а не наоборот. В противном случае велик риск получения в итоге класса, совершенно непригодного или неудобного для использования.

### Проектирование интерфейсной части класса

#### Выбор необходимого уровня абстракции

Для начала попробуем абсолютно забыть о деталях внутренней реализации класса и полностью сосредоточиться на необходимом уровне абстракции сущности «стакан» **в рамках решаемой задачи**.

* Стакан с момента своего создания обладает некоторой **вместимостью**, которая остается неизменной на всем протяжении жизни стакана. Различные стаканы (различные экземпляры класса «Стакан») могут иметь различную вместимость
* **Количество жидкости** в стакане изначально равно нулю
* В стакан можно **добавить** **некоторое** неотрицательное **количество** **воды**. При этом суммарное количество жидкости в стакане не может превышать его вместимость
* Из стакана можно **вылить некоторое** неотрицательное **количество воды**, суммарно не превышающее количество воды, находившееся в стакане.

Итак, после этого небольшого осмысления предметной области мы обнаружили, что наш стакан обладает следующими **свойствами**, определяющими его состояние:

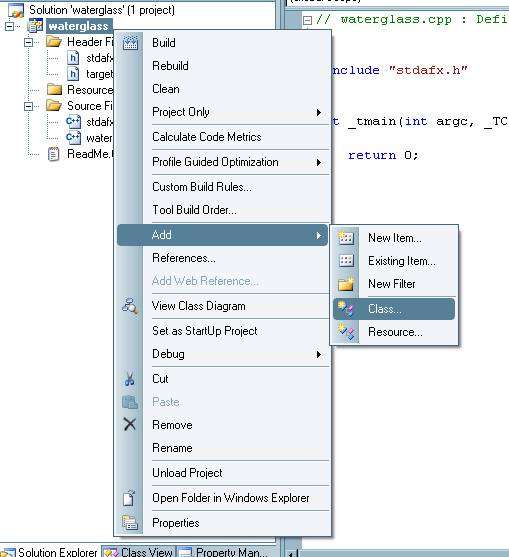
* Вместимость стакана
* Количество жидкости в стакане

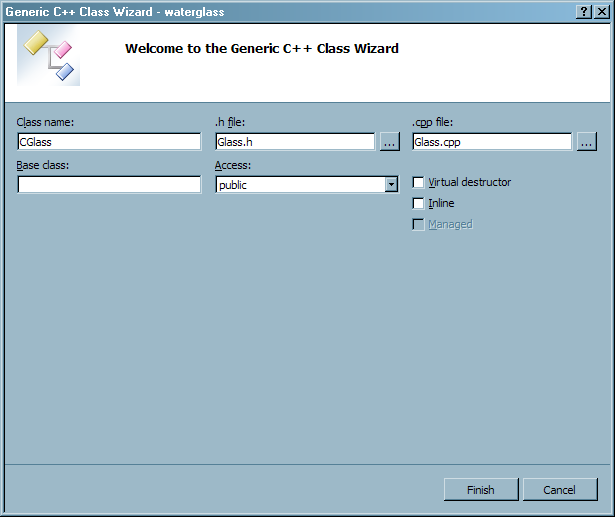
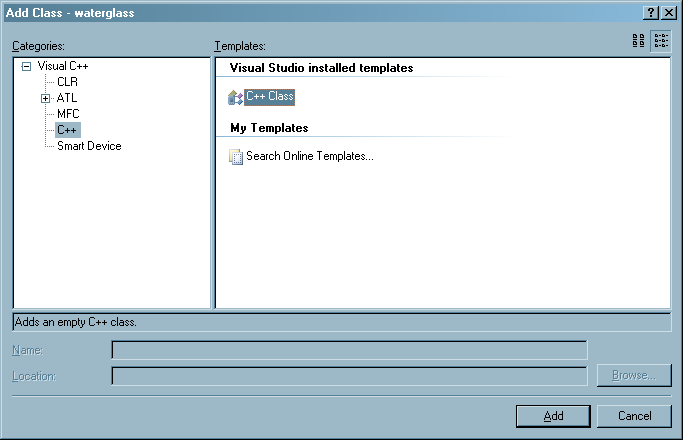
Помимо свойств, над стаканом можно осуществить ряд **действий**:

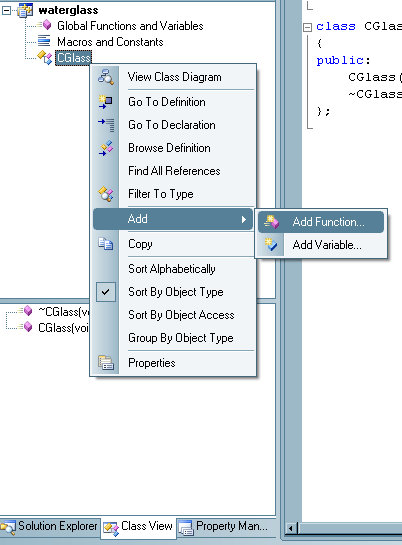
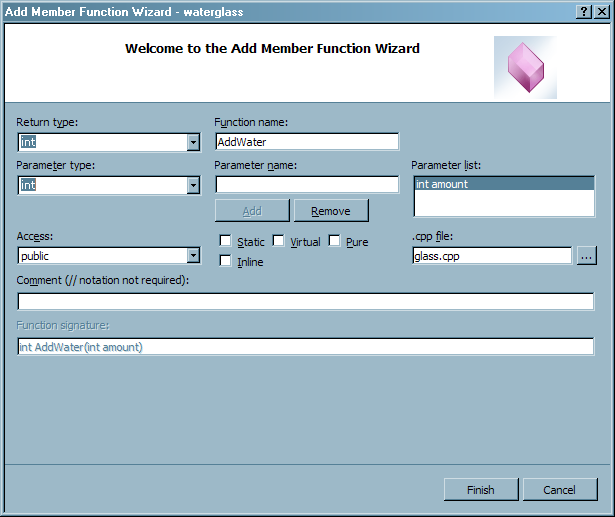
* Добавить воду
* Вылить воду

Действия, совершаемые над объектом, называются **методами**. Их вызов может приводить, а может и не приводить к изменению состояния объекта. Например, действия по выливанию и добавлению воды в стакан приводят к изменению количества жидкости в стакане, изменяя его состояние.

#### Вспомогательные средства Microsoft Visual Studio для добавления класса

Для добавления класса в среде Microsoft Visual Studio можно воспользоваться встроенным мастером добавления нового класса. При помощи контекстного меню Solution Explorer-а выберите пункт Add→Class:  
,  
в открывшемся окне выберите C++ class и укажите имя класса и имена заголовочного файла и файла с реализацией класса:



С помощью контекстного меню панели Class View можно добавить необходимые методы к выбранному классу:  
 

Разумеется, проделать вышеупомянутую работу можно и вручную путем создания и добавления файлов исходного кода к проекту и ручного их редактирования.

#### Запись интерфейсной части класса

Согласно условиям задания определение класса CGlass будет выглядеть следующим образом:

class CGlass

{

public:

// создаем стакан заданной вместимости

CGlass(int capacity);

~CGlass();

// возвращаем вместимость стакана

int GetCapacity()const;

// возвращаем количество воды в стакане

int GetWaterAmount()const;

// добавляем воду в стакан, возвращая фактически добавленное количество

int AddWater(int amount);

// выливаем воду из стакана, возвращаем фактически вылитое количество

int PourWater(int amount);

};

Упомянутые методы класса, а также его конструктор и деструктор, являются публичными и формируют его **интерфейсную часть**.

Отметим, что методы GetCapacity() и GetWaterAmount() возвращают информацию о состоянии стакана, **не изменяя его**, поэтому они должны быть константными. В противном случае, мы не сможем их вызвать для константных экземпляров класса CGlass, а также экземпляров, доступных нам по константной ссылке или указателю на константные данные.

### Разработка через тестирование

На операции добавления и выливания воды из стакана накладывается ряд ограничений, которые необходимо учитывать при реализации класса. При этом программист не застрахован от внесения новых ошибок в ходе исправления старых, а также при дальнейшем развитии класса. Многократное **ручное** тестирование всех методов класса после вносимых в него изменений может отнимать много времени и часто избегается программистами. Это, в свою очередь, может привести к тому, что случайно внесенные ошибки не будут своевременно замечены и устранены. Поэтому часто применяется автоматическое тестирование программ, основанное на разработке вспомогательного тестирующего кода, выполняющего проверку соответствия поведения классов и функций их спецификациям.

Достаточно эффективной техникой программирования является **разработка тестов для класса до его реализации** или параллельно с нею, управляя при этом его разработкой. Данный подход к разработке кода называется «**Test-driven development**» ([разработка через тестирование](http://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0_%D1%87%D0%B5%D1%80%D0%B5%D0%B7_%D1%82%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), **TDD**) и способствует разработке надежного и устойчивого к сбоям кода.

Кроме того, если класс разрабатывается с учетом его тестирования отдельно от других классов программы, то он получается легче в использовании и дальнейшем сопровождении, т.к. содержит меньше зависимостей от внешнего кода - чем больше зависимостей в классе от внешних условий, тем сложнее написать для него тесты.

Разработка же тестов для уже написанного класса обладает меньшей эффективностью по целому ряду причин.

Во-первых, такие тесты составляются менее тщательно, т.к., по мнению программиста, «цель уже достигнута» и такая работа кажется обузой. При TDD, напротив, составление тестов является естественной составляющей процесса разработки.

Во-вторых, когда класс уже разработан, разработка тестов для него затрудняется его зависимостями от других классов, которыми класс успел обрасти в процессе разработки. Устранение таких зависимостей, необходимое для нормального тестирования класса, зачастую связано с необходимостью изменения архитектуры класса (**рефакторинга**), в процессе которого существует вероятность сломать уже «работающий» по мнению программиста код при отсутствии средств автоматического контроля его работы. Разработка через тестирование, напротив, поощряет внесение изменений в код, поскольку качественно написанные тесты значительно повышают вероятность обнаружения ошибок в процессе рефакторинга.

В конце концов, программист может вообще прийти к мысли, что разработка тестов для его кода – не барское занятие, что код он уже 100 раз перепроверил, и ошибок в нем нет. Тем не менее, даже если ошибок сегодня в коде действительно нет (что случается очень редко), существует риск внести ошибку в дальнейшем, например, при наращивании функциональности класса. При TDD ошибка при модификации кода может быть обнаружена с большей вероятностью.

Для проверки работоспособности методов класса **еще до их полноценной реализации** напишем несколько тестов в основном файле программы:

#include "stdafx.h"

#include "Glass.h"

#include <assert.h>

using namespace std;

void TestGlassCreation()

{

// проверяем свойства созданного стакана

CGlass validGlass(10);

assert(validGlass.GetCapacity() == 10);

assert(validGlass.GetWaterAmount() == 0);

// пытаемся создать скакан отрицательной вместимости

CGlass invalidGlass(-10);

assert(invalidGlass.GetCapacity() == 0);

assert(invalidGlass.GetWaterAmount() == 0);

cout << "Glass creation test is ok\n";

}

void TestAddingWater()

{

// создаем стакан вместимостью 100 мл.

CGlass someGlass(100);

// добавляем в него воды по 1 мл, убеждаясь в невозможности переполнения

for (int i = 1; i < 200; ++i)

{

const int addedWater = someGlass.AddWater(1);

if (i <= someGlass.GetCapacity())

{

assert(addedWater == 1);

assert(someGlass.GetWaterAmount() == i);

}

else

{

assert(addedWater == 0);

assert(someGlass.GetWaterAmount() == someGlass.GetCapacity());

}

}

CGlass anotherGlass(100);

anotherGlass.AddWater(10);

assert(anotherGlass.GetWaterAmount() == 10);

// пытаемся добавить отрицательное количество воды

const int addedWater = anotherGlass.AddWater(-10);

assert(addedWater == 0);

assert(anotherGlass.GetWaterAmount() == 10);

cout << "Adding water test is ok\n";

}

void TestPouringWater()

{

// создаем стакан

CGlass glass(100);

// наполняем его доверху

glass.AddWater(glass.GetCapacity());

assert(glass.GetWaterAmount() == glass.GetCapacity());

// выливаем воду по 1 мл, убеждаясь в невозможности вылить больше, чем есть воды

for (int i = glass.GetWaterAmount() - 1; i >= -100; --i)

{

const int pouredWater = glass.PourWater(1);

if (i >= 0)

{

assert(pouredWater == 1);

assert(glass.GetWaterAmount() == i);

}

else

{

assert(pouredWater == 0);

assert(glass.GetWaterAmount() == 0);

}

}

// создаем другой стакан

CGlass anotherGlass(100);

anotherGlass.AddWater(10);

assert(anotherGlass.GetWaterAmount() == 10);

// пытаемся добавить отрицательное количество воды

const int pouredWater = anotherGlass.PourWater(-10);

assert(pouredWater == 0);

assert(anotherGlass.GetWaterAmount() == 10);

cout << "Pouring water test is ok\n";

}

void TestIntegerOverflow()

{

CGlass glass(100);

glass.AddWater(10);

assert(glass.GetWaterAmount() == 10);

// пытаемся вызвать целочисленное переполнение

const int addedWater = glass.AddWater(INT\_MAX);

assert(addedWater == glass.GetCapacity() - 10);

assert(glass.GetWaterAmount() == glass.GetCapacity());

CGlass anotherGlass(100);

// пытаемся вызвать целочисленное переполнение

const int pouredWater = anotherGlass.PourWater(INT\_MAX);

assert(pouredWater == 0);

assert(anotherGlass.GetWaterAmount() == 0);

cout << "Integer overflow test is ok\n";

}

int main(int argc, char\* argv[])

{

TestGlassCreation();

TestAddingWater();

TestPouringWater();

TestIntegerOverflow();

return 0;

}

Для того чтобы тесты проходили компиляцию, необходимо в теле методов класса и конструкторах поместить заглушки.

Теперь, когда с интерфейсной частью класса мы определились, пришло время определиться с его реализацией.

Для хранения информации о вместимости стакана и количестве находящейся в нем воды нам понадобятся соответствующие целочисленные переменные. Для сокрытия подробностей реализации класса разместим члены-данные класса в закрытой области объявления класса:

class CGlass

{

public:

// создаем стакан заданной вместимости

CGlass(int capacity);

~CGlass();

// возвращаем вместимость стакана

int GetCapacity()const;

// возвращаем количество воды в стакане

int GetWaterAmount()const;

// добавляем воду в стакан, возвращая фактически добавленное количество

int AddWater(int amount);

// выливаем воду из стакана, возвращаем фактически вылитое количество

int PourWater(int amount);

private:

int m\_capacity; // вместимость

int m\_waterAmount; // количество воды

};

Для отличия имен переменных-членов класса от локальных переменных и параметров его методов мы используем префикс **m\_** для их именования.

В файле Glass.cpp напишем реализацию методов и конструктора. Их разработка во многом облегчилась благодаря предварительно написанным тестам.

#include "StdAfx.h"

#include "Glass.h"

// создаем пустой стакан заданной неотрицательной вместимости

CGlass::CGlass(int capacity)

:m\_capacity((capacity >= 0) ? capacity : 0)

,m\_waterAmount(0)

{

}

CGlass::~CGlass()

{

}

// возвращаем вместимость стакана

int CGlass::GetCapacity()const

{

return m\_capacity;

}

// возвращаем количество воды в стакане

int CGlass::GetWaterAmount()const

{

return m\_waterAmount;

}

// добавляем воду в стакан, возвращая фактически добавленное количество

int CGlass::AddWater(int amount)

{

if (amount < 0)

{

amount = 0;

}

// проверка на целочисленное переполнение при сложении и на переполнение стакана

else if (

(m\_waterAmount + amount < m\_waterAmount) ||

(m\_waterAmount + amount > m\_capacity))

{

amount = m\_capacity - m\_waterAmount;

}

m\_waterAmount += amount;

return amount;

}

// выливаем воду из стакана, возвращаем фактически вылитое количество

int CGlass::PourWater(int amount)

{

// нельзя вылить больше, чем имеется в наличии, а также отрицательное количество

amount =

(amount < 0) ? 0 :

(m\_waterAmount >= amount) ? amount : m\_waterAmount;

m\_waterAmount -= amount;

return amount;

}

### Резюме

Вы получили основное представление об объектно-ориентированном подходе к моделированию сущностей предметной области, о том, как создавать собственные классы на языке C++, а также вести разработку с использованием подхода «Test driven development».

## Практические задания

На оценку «**удовлетворительно**» необходимо набрать **не менее 50 баллов**.

На оценку «**хорошо**» набрать **не менее 300 баллов**.

На оценку «**отлично**» набрать **не менее 600 баллов**.

**Внимание, дополнительные задания принимаются только после успешной защиты обязательных заданий**.

### Обязательные задания

#### Задание 1

Выполните один из предложенных вариантов задания.

##### Вариант 1 – Телевизор – 50 баллов

Спроектируйте **с использованием TDD или BDD** класс CTVSet, моделирующий телевизор, который может находиться либо в выключенном, либо включенном состоянии. Находясь во включенном состоянии, телевизор способен отображать один из 99 каналов (от 1 до 99).

Над телевизором можно выполнять следующие действия:

* Выключить. В выключенном состоянии нельзя переключать каналы.
* Включить. При своем включении телевизор включается на том канале, на котором он был ранее выключен. При самом первом включении телевизор включается на первом канале.
* Выбрать заданный канал (от 1 до 99) или остаться на том же самом канале, если номер канала за пределами данного диапазона.

Телевизор обладает следующими свойствами:

* Включен или выключен
* Номер текущего канала. В выключенном состоянии номер текущего канала должен быть равен нулю.

Разработайте программу, использующую разработанный Вами класс CTVSet, которая обрабатывает команды пользователя, вводимые им со стандартного потока ввода:

* **TurnOn**. Включает телевизор, если он был выключен
* **TurnOff**. Выключает телевизор, если он был включен
* **SelectChannel** *<номер канала>*. Выбирает указанный номер канала.
* **Info**. Выводит текущее состояние телевизора (выключен или включен, номер канала).

###### Бонус в 10 баллов за возможность переключения на ранее просмотренный канал

Бонус присуждается за добавление в класс телевизора метода SelectPreviousChannel(), выполняющей переключение телевизора на канал, на котором телевизор находился ранее, либо остаться на том же канале. После выключения телевизора информация о ранее предыдущем канале должна сохраняться. Данная операция также доступна только у включенного телевизора.

Интерфейс пользователя должен поддерживать соответствующую команду, вводимую с stdin.

###### Бонус в 40 баллов за возможность задать имя для канала

Реализовать возможность задания имени для канала с допустимым номером при помощи метода **SetChannelName**(номер канала, имя). Именем канала может являться непустая строка. Имя канала может содержать пробелы, но не должно состоять из одних только пробелов. Пробелы в начале и в конце имени канала, а также лишние пробелы в середине должны игнорироваться.

Добавить версию метода **SelectChannel**, принимающую строку, задающую имя канала.

Должна иметься возможность удалить имя канала при помощи метода **DeleteChannelName**(имя канала), узнать имя канала по его порядковому номеру при помощи метода **GetChannelName**(номер канала), а также получить номер канала по его имени при помощи метода **GetChannelByName**(имя канала).

Все эти операции доступны только у **включенного** телевизора.

Если указанное имя уже было закреплено за другим каналом, то происходит смена ассоциации. Если за текущим каналом ранее было закреплено другое имя, то старая ассоциация должна быть удалена. Иными словами, с каждым каналом может быть связано не более одного имени. Одному имени может соответствовать только один канал.

Интерфейс пользователя предоставлять возможность задавать и удалять имена каналов, переключать каналы при помощи SelectChannel не только по номеру, но и имени. Обновить формат вывода команды Info, чтобы выводился еще и список каналов, с которыми ассоциированы имена, отсортированный в порядке возрастания номера канала:

1 – ОРТ

17 – Discovery

35 – Русский экстрим

##### Вариант 2 – Автомобиль – 140 баллов

Разработать класс Car, моделирующий автомобиль в следующей предметной области. Для разработанного класса должны быть также разработаны модульные тесты.

Двигатель автомобиля может находиться как во включенном состоянии, так и в выключенном.

В автомобиле может быть включена одна из следующих передач:

* Задний ход (-1)
* Нейтральная передача (0)
* Первая передача (1)
* Вторая передача (2)
* Третья передача (3)
* Четвертая передача (4)
* Пятая передача (5)

Каждая передача автомобиля имеет свой диапазон скоростей:

|  |  |
| --- | --- |
| **Передача** | **Диапазон скоростей** |
| Задний ход | 0 – 20 |
| Нейтраль | Без ограничений |
| Первая | 0 – 30 |
| Вторая | 20 – 50 |
| Третья | 30 – 60 |
| Четвертая | 40 – 90 |
| Пятая | 50 – 150 |

На каждой передаче можно развить скорость в пределах отведенного данной передаче диапазона. **Исключение – нейтральная передача**, на которой скорость можно изменить **только уменьшить (по модулю)**.

**При включенном двигателе** переключиться можно с любой передачи на любую при условии, что текущая скорость автомобиля находится в диапазоне скоростей новой передачи, и направление движения автомобиля допускает включение данной передачи. Например:

* на задний ход можно переключиться только **на нулевой скорости**; даже при движении назад на нейтральной передаче не допускается включать заднюю передачу.
* с заднего хода можно переключиться на первую передачу **только на нулевой скорости;**
* переключившись на заднем ходу на нейтральную передачу на ненулевой скорости, переключиться на переднюю передачу **можно только после остановки**

Двигатель данного автомобиля может быть **выключен** только при **нулевой** скорости на **нейтральной** передаче. При выключенном двигателе переключиться можно только на нейтральную передачу. Как следствие, автомобиль **после включения двигателя** находится на **нейтральной** передаче **в состоянии покоя**.

Автомобиль обладает следующими свойствами:

* Состояние двигателя (включен или выключен) – метод IsTurnedOn()
* Направление движения (вперед, назад или стоим на месте) – метод GetDirection()
* Текущая скорость движения (целое число от 0 до максимальной скорости) – метод GetSpeed()
* Текущая выбранная передача [-1..5] – метод GetGear()

Автомобиль может выполнять следующие действия:

* Включить двигатель (если он выключен). Возвращает true, если двигатель включился (или уже был включен).  
  bool **TurnOnEngine**()
* Выключить двигатель (если он включен и текущая передача – нейтральная, а автомобиль стоит). Возвращает true, если двигатель был успешно выключен (или уже был выключен), и false, если двигатель не может быть в данный момент выключен.  
  bool **TurnOffEngine**()
* Выбрать указанную передачу [-1..5]. В случае успешного переключения передачи (в том числе и на саму себя) возвращает true.  
  bool **SetGear**(int gear)
* Задать указанную скорость. Возвращает true, если скорость удалось изменить и false, если изменить скорость движения на указанную невозможно (например, на нейтральной передаче нельзя разогнаться).  
  bool **SetSpeed**(int speed)

На основе данного класса разработать приложение, позволяющее пользователю управлять виртуальным автомобилем при помощи следующих команд, вводимых со стандартного потока ввода:

* **Info**. Выводит состояние двигателя автомобиля, направление движения, скорость и передачу
* **EngineOn**. Включает двигатель
* **EngineOff**. Выключает двигатель
* **SetGear *передача***. Включает заданную передачу, заданную числом от -1 до 5. В случае ошибки сообщает о причине невозможности переключения передачи
* **SetSpeed *скорость***. Устанавливает указанную скорость движения, заданную неотрицательным числом. В случае невозможности изменения скорости сообщает о причине невозможности изменить скорость на указанную.

##### Вариант 3 – Прямоугольник - 50 баллов

Разработайте **с использованием TDD** класс CRectangle, моделирующий сущность «Прямоугольник».

Прямоугольник обладает следующими свойствами[[1]](#footnote-1) (не путать с членами-данными[[2]](#footnote-2)):

* Ширина (width), доступна как для чтения, так и для записи
* Высота (height), доступна как для чтения, так и для записи
* Координата X левого края (Left), доступна как для чтения, так и для записи
* Координата Y верхнего края (Top), доступна как для чтения, так и для записи
* Координата X правого края (Right), доступна как для чтения, так и для записи
* Координата Y нижней стороны (Bottom), доступна как для чтения, так и для записи
* Площадь (Area), доступна только для чтения
* Периметр (Perimeter), доступен только для чтения

Координаты и размеры прямоугольника **задаются целыми числами**.

Ширина и высота

Размеры (ширина и высота) прямоугольника не могут быть отрицательными (конструкторе и set-методах следует заменять отрицательные размеры на 0).

x

y

left

right

top

bottom

width

height

Конструктор данного класса принимает координаты левого верхнего угла прямоугольника, а также его ширину и высоту.

Над прямоугольником можно выполнить следующие действия:

* Перенести вдоль заданного вектора на dx и dy, не изменяя размеров прямоугольника  
  **void Move(int dx,int dy)**
* Отмасштабировать прямоугольник с использованием масштабных коэффициентов sx и sy  
  **void Scale(int sx, int sy)**  
  При масштабировании координаты левого верхнего угла прямоугольника остаются без изменения, а изменяется только его размер.  
  Если sx или sy является отрицательным числом, то масштабирования не происходит
* Найти пересечение данного прямоугольника с другим прямоугольником:  
  **bool Intersect(CRectangle const& other)**  
  Данный метод возвращает true, если прямоугольники пересекаются, и изменяет характеристики текущего прямоугольника. Если прямоугольники не пересекаются, то данный метод возвращает false и сбрасывает ширину и высоту прямоугольника в 0 (координаты верхнего левого угла остаются без изменений).

Разработать на основе данного класса программу, выполняющую считывание двух текстовых файлов, имена которых передаются программе через параметры командной строки, содержащих произвольное количество команд (по одной команде в каждой строке), управляющих размерами и трансформациями двух прямоугольников.

* **Rectangle <*left> <top> <width> <height>***. Инициализирует текущий прямоугольник указанными координатами и размерами.
* **Move *<dx> <dy>***. Переносит текущий прямоугольник вдоль заданного вектора, не изменяя его размеров
* **Scale *<sx> <sy>***. Масштабирует (увеличивает в указанное количество раз ширину и высоту) текущий прямоугольник с использованием масштабных коэффициентов относительно его верхнего левого угла

До первого появления команды Rectangle в файле текущий прямоугольник **имеет нулевые размеры и нулевые координаты верхнего левого угла**.

После выполнения команд, задаваемых в текстовых файлах программа должна вывести в стандартный поток вывода координаты и размеры прямоугольников после применения указанных преобразований, их периметр и площадь, а также результат их пересечения в следующем формате:

**Rectangle 1:**

**Left top: (*<left1>; <top1>)***

***Size: <width1>\*<height1>***

***Right bottom:* (*<right1>; <bottom1>)***

***Area: <area1>***

***Perimeter: <perimeter1>***

**Rectangle 2:**

**Left top: (*<left2>; <top2>)***

***Size: <width2>\*<height2>***

***Right bottom:* (*<right2>; <bottom2>)***

***Area: <area2>***

***Perimeter: <perimeter2>***

**Intersection rectangle:**

**Left top: (*<left>; <top>)***

***Size: <width>\*<height>***

***Right bottom:* (*<right>; <bottom>)***

***Area: <area>***

***Perimeter: <perimeter>***

###### Бонус в 50 баллов за визуализацию исходных прямоугольников и результата их пересечения

Разработайте класс CCanvas, моделирующий прямоугольное растровое полотно для рисования в текстовом режиме, а также метод FillRectangle, выполняющий закрашивание области, соответствующей прямоугольнику на данном полотне.

Каркас класса CCanvas представлен ниже:

/\*

Класс, моделирующий сущность Canvas (полотно, холст для рисования, картинка, канва),

хранящую прямоугольный массив пикселей. Для каждого пикселя изображения можно задать

свой код символа, что позволяет выводить простейшие картинки в текстовом режиме,

вроде таких:

+----+

/ /|

+----+ |

| | +

| |/

+----+

\*/

class CCanvas

{

public:

// Создает канву для рисования размером width\*height

// После своего создания содержимое канвы заполнено пробельными символами

// Допускается создание канвы нулевых размеров

CCanvas(unsigned width, unsigned height);

// Возвращает ширину канвы

unsigned GetWidth()const;

// Возвращает высоту канвы

unsigned GetHeight()const;

// Очищает канву (заполняет содержимое символами с указанным кодом)

// Если код символа находится в диапазоне от 0 до ' ', команда игнорируется

void Clear(char code = ' ');

// Задает код символа code для пикселя в координатах (x, y)

// Координаты верхнего левого угла канвы принимаются равными 0, 0.

// Если координаты выходят за пределы канвы, либо код символа

// находится в диапазоне от 0 до (' ' - 1), содержимое канвы не должно меняться

void SetPixel(int x, int y, char code);

// Возвращает код символа пикселя в координатах (x, y)

// Координаты верхнего левого угла канвы принимаются равными 0, 0.

// Если координаты пикселя выходят за пределы канвы, должен возвращаться

// код символа "пробел"

char GetPixel(int x, int y)const;

// Выводит содержимое в поток вывода, производный от std::ostream

// (например, std::cout, экземпляр ofstream, или ostringstream)

// В конце каждой строки должен выводиться символ \n

void Write(std::ostream & ostream)const;

private:

// Закрытые данные и методы класса

};

Объявление функции FillRectangle представлено ниже

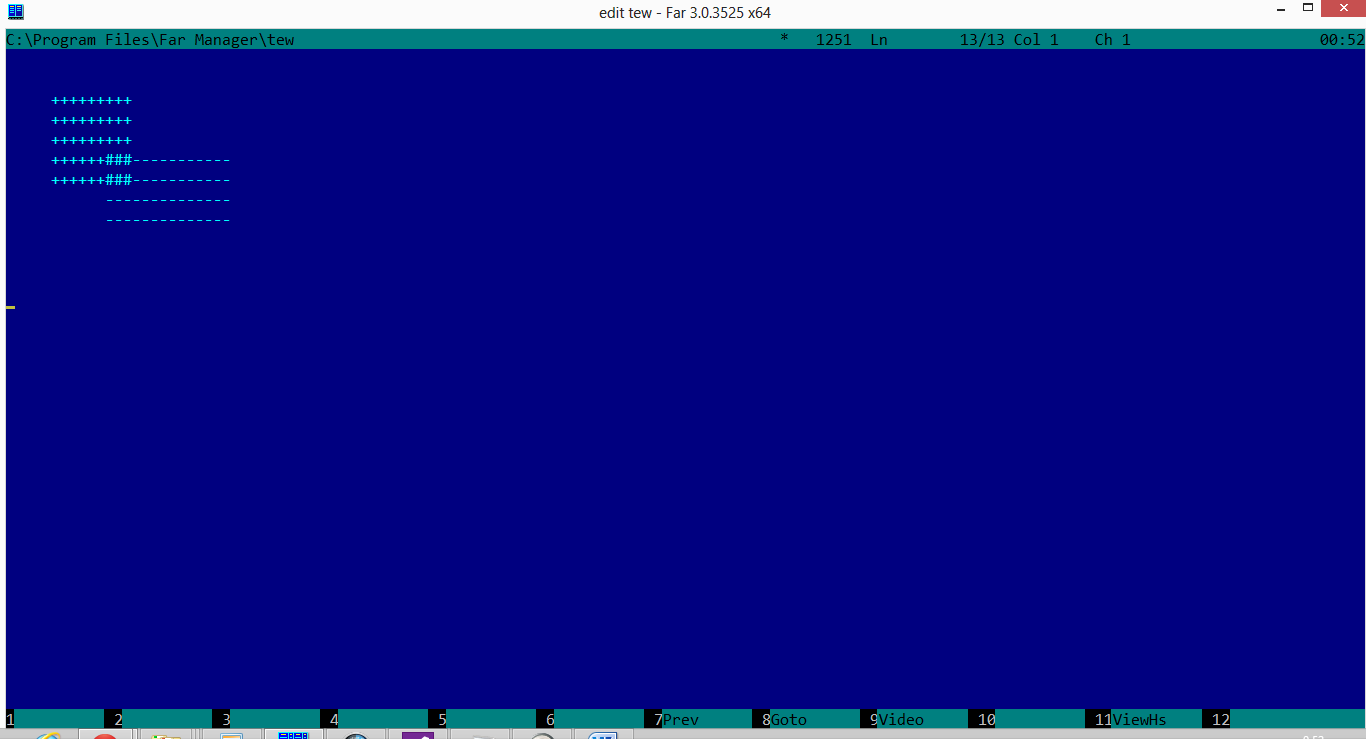
// Заполняет пиксели, соответствующие прямоугольнику rect в канве,

// символом с кодом code

void FillRectangle(CRectangle const& rect, char code, CCanvas & canvas);

Первый прямоугольник должен быть отображен при помощи символов “+”, второй – при помощи символа “-“, а результат их пересечения – при помощи символа “#”. На следующем рисунке показан ожидаемый результат работы программы при следующих входных данных:

* Прямоугольник 1. Left: 5, Top: 2, Width: 9, Height: 5
* Прямоугольник 2. Left: 11, Top: 5, Width: 14, Height: 4



Программа должна вывести визуальный результат пересечения прямоугольников в файл, имя которого передано в качестве 3-го аргумента командной строки, а при отсутствии 3-го аргумента – в стандартный поток вывода (сразу после текстовой информации, указанной в обязательной части задания). В обоих случаях принят размер полотна для рисования равным 60 \* 20.

### Дополнительные задания

#### Задание 2 – Калькулятор – 300 баллов

Разработать **с использованием подхода TDD** программу «Калькулятор», позволяющую выполнять простейшие арифметические операции над переменными и пользовательскими функциями. Спроектируйте классы, моделирующих основные сущности предметной области, и напишите для них тесты.

##### Формат входных и выходных данных

Входные данные поступают из stdin. Работа программы завершается при обнаружении конца файла. Каждая строка содержит одну из следующих команд:

* **var <*идентификатор*>**  
  **Объявляет** **переменную** типа double с именем <*идентификатор*>. Идентификатор **не должен совпадать** ни с одним из **ранее объявленных** имен переменных и функций. Значение переменной после ее объявления не определено (можно использовать значение NAN для этих целей). В отсутствие ошибок в stdout ничего не выводится. При наличии ошибки команда игнорируется, f в stdout выводится текст ошибки.
* **let <*идентификатор1*> = <*число с плавающей запятой*>** либо  
  **let <*идентификатор1*> = <*идентификатор2*>**  
  Присваивает **переменной** с именем <*идентификатор1*> числовое значение, либо **текущее значение** **ранее объявленного** идентификатора с именем <*идентификатор2*>. Если переменная с именем <*идентификатор1*> не была ранее объявлена, происходит объявление новой переменной. В качестве <*идентификатора1*> не может выступать имя функции. В отсутствие ошибок в stdout ничего не выводится. В случае ошибки команда игнорируется, а в stdout выводится текст ошибки.
* **fn <*идентификатор1*> = <*идентификатор2*>** либо  
  **fn <*идентификатор1*> = <*идентификатор2*>[<*операция*> <*идентификатор3*>**  
  Объявляет новую функцию с ранее **необъявленным именем** <*идентификатор1*>, значением которой будет либо значение идентификатора <*идентификатор2*>, либо результат применения одной из следующих бинарных *операций* к значениям **ранее объявленных** идентификаторов <*идентификатор2*> и <*идентификатор3*> **в момент вычисления значения функции**:
  + +. Сложение.
  + -. Вычитание.
  + \*. Умножение
  + /. Деление

Если значение хотя бы одного из операндов операции не определено, результатом операции должно быть неопределенное значение. В отсутствие ошибок в stdout ничего не выводится. В случае ошибки команда игнорируется, а в stdout выводится текст ошибки.

* **print <*идентификатор*>**  
  Выводит в stdout значение **ранее объявленного** идентификатора. Если идентификатором являлась переменная, то выводится ее значение, а если функция, то выводится вычисленное значение функции. **Значение идентификатора выводится с точностью в 2 знака** после запятой. В случае, когда значение идентификатора не определено, должно быть выведено **nan**. В случае ошибки (например, попытка вывести значение необъявленного идентификатора), команда игнорируется, а в stdout должен быть выведен текст ошибки.
* **printvars**  
  Выводит в stdout имена и значения всех ранее объявленных переменных, **отсортированных по алфавиту**, по одному в каждой строке в следующем формате:  
  <идентификатор>:<значение>  
  Значение переменной выводится с точностью **2 знака после запятой**. Если значение переменной не определено, должно быть выведено **nan**. Если ни одной функции не было объявлено к моменту выполнения команды printvars, в stdout выводиться ничего не должно.
* **printfns**  
  Выводит в stdout имена и значения всех ранее объявленных функций, **отсортированных по алфавиту**, по одному в каждой строке в следующем формате:  
  **<идентификатор>:<значение>**  
  Значение функции выводится с точностью **в 2 знака после запятой**. Если значение функции не определено, должно быть выведено **nan**. Если ни одной функции не было объявлено к моменту выполнения команды printfns, в stdout выводиться ничего не должно

**Идентификатором** является непустая строка, в которой допускается использовать буквы английского алфавита, цифры и символ подчеркивания. Идентификатор не может начинаться с цифры. Идентификаторы используются в качестве имен переменных и функций.

Идентификаторы и названия команд являются **чувствительными к регистру символов**.

##### Примеры

###### Объявление, присваивание и вывод значений переменных

|  |  |  |
| --- | --- | --- |
| **stdin** | **stdout** | **Пояснение** |
| var x |  |  |
| print x | nan | Значение переменной **x** пока не определено |
| let x=42 |  | Присваиваем переменной x значение 42 |
| print x | 42.00 |  |
| let x=1.234 |  | Значение переменной можно изменить |
| print x | 1.23 | Значение выводится с точность 2 знака после запятой |
| let y=x |  | Автоматически объявляем переменную **y** и присваиваем ей текущее значение **x** |
| let x=99 |  |  |
| printvars | x:99.00  y:1.23 | Переменная **y** хранит присвоенное ей значение **x**. Последующие манипуляции над x не оказывают на нее влияния. |

###### Объявление функций

|  |  |  |
| --- | --- | --- |
| **stdin** | **stdout** | **Пояснение** |
| var x |  |  |
| var y |  |  |
| fn XPlusY=x+y |  |  |
| print XPlusY | nan | Значение функции не определено, т.к. не определены значения ее аргументов |
| let x=3 |  |  |
| let y=4 |  |  |
| print XPlusY | 7.00 | Теперь значение функции определено |
| let x=10 |  |  |
| print XPlusY | 14.00 | Значение функции зависит от значений ее аргументов |
| let z=3.5 |  |  |
| fn XPlusYDivZ=XPlusY/z |  | Значение функции может зависеть не только от значений переменных, но и от значений других функций |
| printfns | XPlusY:14.00  XPlusYDivZ:4.00 |  |

###### Еще раз про различие между fn и let

|  |  |  |
| --- | --- | --- |
| **Stdin** | **stdout** | **Пояснение** |
| let v=42 |  |  |
| let variable=v |  | variable хранит значение v (42) |
| fn function=v |  | function хранит действие, которое будет вычислено при получении значения функции |
| let v=43 |  |  |
| print variable | 42.00 |  |
| print function | 43.00 | Значением function будет значение переменной v, вычисленное в момент вызова функции (а не ее объявления) |

###### Вычисление площади круга

|  |  |  |
| --- | --- | --- |
| **stdin** | **stdout** | **Пояснение** |
| var radius |  |  |
| let pi=3.14159265 |  |  |
| fn radiusSquared=radius\*radius |  |  |
| fn circleArea=pi\*radiusSquared |  |  |
| let radius=10 |  |  |
| print circleArea | 314.16 |  |
| let circle10Area=circleArea |  | Circle10Area хранит значение функции circleArea, вычисленной при radius=10 |
| let radius=20 |  |  |
| let circle20Area=circleArea |  | circle20Area хранит значение функции circleArea, вычисленной при radius=20 |
| printfns | circleArea:1256.64  radiusSquared:400.00 |  |
| printvars | circle10Area:314.16  circle20Area:1256.64  pi:3.14  radius:20.00 |  |

###### Вычисление последовательности Фибоначчи

|  |  |  |
| --- | --- | --- |
| **stdin** | **stdout** | **Пояснение** |
| let v0=0 |  |  |
| let v1=1 |  |  |
| fn fib0=v0 |  |  |
| fn fib1=v1 |  |  |
| fn fib2=fib1+fib0 |  |  |
| fn fib3=fib2+fib1 |  |  |
| fn fib4=fib3+fib2 |  |  |
| fn fib5=fib4+fib3 |  |  |
| fn fib6=fib5+fib4 |  |  |
| printfns | fib0:0.00  fib1:1.00  fib2:1.00  fib3:2.00  fib4:3.00  fib5:5.00  fib6:8.00 |  |
| let v0=1 |  |  |
| let v1=1 |  |  |
| printfns | fib0:1.00  fib1:1.00  fib2:2.00  fib3:3.00  fib4:5.00  fib5:8.00  fib6:13.00 |  |

##### Бонус в 200 баллов за оптимизацию вычислений функций

Наивный подход к вычислению значений функций может в ряде случаев приводить к экспоненциальной вычислительной сложности. Например, попытка вычислить значение хотя бы 50-го числа последовательности Фибоначчи, заданного в виде «рекурсивной» последовательности функций, потребует весьма продолжительного времени. Придумайте способ, позволяющий значительно сократить вычислительную сложность алгоритма в таких ситуациях.

##### Бонус в 100 баллов за возможность вычисления очень больших последовательностей функций

Использование рекурсии в процессе вычислений функций с большой глубиной вычислений (десятки или сотни тысяч) может привести к переполнению стека (win32-приложениям по умолчанию доступно около 2МБ стека). Придумайте способ решить проблему с переполнением стека из-за глубокой рекурсии, не изменяя размер стека в настройках компоновщика.

Пример входных данных, которые потенциально могут приводить к указанной проблеме:

|  |  |  |
| --- | --- | --- |
| **stdin** | **stdout** | **Пояснение** |
| let x=1  fn x2=x+x  fn x3=x2+x  fn x4=x3+x  fn x5=x3+x  …  fn x1000000=x999999+x |  | Объявляется большая последовательность функций, каждая из которых прибавляет x к значению предыдущей функции. |
| print x1000000 | 1000000.00 | Наивное рекурсивное вычисление функции x1000000 может привести к переполнению стека. |
| let x=2 |  |  |
| print x1000000 | 2000000.00 |  |

## Ссылки

1. [Разработка через тестирование](http://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0_%D1%87%D0%B5%D1%80%D0%B5%D0%B7_%D1%82%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)

1. Т.к. в C++ отсутствует поддержка свойств, следует вместо них использовать соответствующие Get\* и Set\* методы. [↑](#footnote-ref-1)
2. Используйте тот набор переменных-членов класса, который необходим и достаточен для реализации упомянутых свойств [↑](#footnote-ref-2)