**Введение в модульное тестирование в Visual Studio C++**

**Общие понятия**

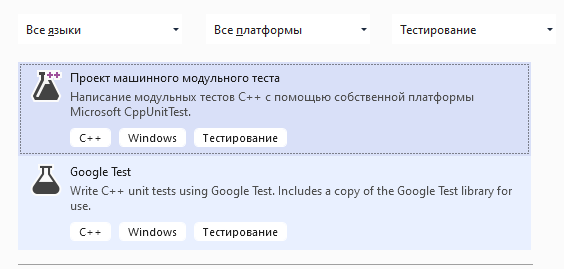
**Модульное тестирование** или **юнит-тестирование** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *unit testing*), иногда **блочное тестирование** — проверка корректности работы отдельных частей исходного кода программы. Под модулем в этом контексте понимаются не столько собственно *модули*, сколько функции (подпрограммы), методы, классы и другие единицы организации программы.

Тестирование проводится путем сверки результата, выдаваемого функцией или методом при различных значениях входных параметров и состояниях объекта, с ожидаемым результатом. Даже простой запуск собранного приложения на соответствие результата с ожиданием может рассматриваться как модульное тестирование, но типично модульное тестирование проводится автоматически: задаются наборы входных данных и ожидаемых при них выходных данных, сверка результата производится компьютером.

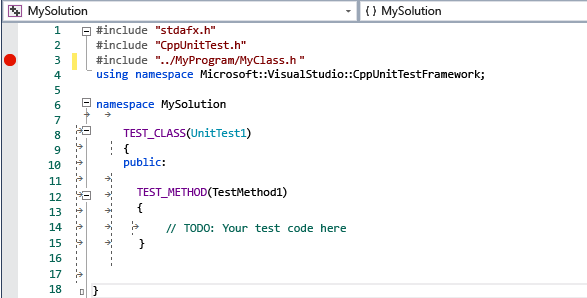
Идея модульного тестирования состоит в том, что параллельно основному компоненту программы, который включает непосредственно алгоритмы ее работы, создается дополнительный «тестовый», в котором имитируется работа основного компонента в тех или иных условиях. По результатам выполнения «тестового» компонента судят о правильности работы основного.

В настоящее время для создания подобных сценариев нет необходимости разрабатывать какие-либо сложные технические решения. Существует масса готовых фреймворков, которые не только облегчают разработку тестов, но и берут на себя значительную часть работы по анализу и представлению их результатов.

Подобные фреймворки часто входят в состав интегрированных сред разработки (IDE). Собственный фреймворк для модульных тестов имеет и Visual Studio.



В CPP-файле в тестовом проекте определены класс-заглушка и метод-заглушка. Они служат примером того, как следует писать код теста. В сигнатурах используются макросы TEST\_CLASS и TEST\_METHOD, что позволяет обнаруживать методы в окне обозревателя тестов.



Тестирующий метод обычно содержит три необходимых компонента:

1. исходные данные: входные значения и ожидаемый результат;
2. код, вычисляющий значение с помощью тестируемого метода;
3. код, сравнивающий ожидаемый результат с полученным.

Кроме этого стоит отметить, что отдельно взятый юнит-тест должен обладать следующими свойствами:

* повторяем — в результате запуска тест на выходе всегда выдает одно и то же значение (всегда приводит систему в одно и то же состояние);
* стабилен — в какое бы время дня и ночи тест бы не запускался, он либо всегда проходит, либо всегда не проходит;
* изолирован — порядок запуска всех имеющихся юнит-тестов, а также действия, выполняемые внутри тестов, никак не влияют на результат выполнения отдельно взятого юнит-теста.

**Примеры**

Рассмотрим процесс создания юнит теста на примерах:

**Пример 1**. Написать функцию, которая по входным данным строит вектор, содержащий координаты начала и конца отрезков и соответствующий UnitTest

Определим типы данных:

typedef pair <double, double> point;

typedef vector < pair< point, point > > vec;

Поскольку данные будем вводить с клавиатуры, то параметром функции сделаем поток ввода, а результат – построенный вектор.

vec create(stringstream& l\_cin) {

vec rez;

while( ! l\_cin.eof()) {

double x1, y1, x2, y2;

l\_cin >> x1 >> y1 >> x2 >> y2;

rez.push\_back(make\_pair(make\_pair(x1, y1), make\_pair(x2, y2)));

}

return rez;

}

Для ручного тестирования в функции main() вводим данные и формируем поток ввода для функции:

int main() {

int k;

cin >> k;

stringstream ss;

double a, b, c, d;

for (int i = 0; i < k-1; i++) {

cin >> a >> b >> c >> d; // вводим координаты начала и конца

// отрезка с клавиатуры

// формируем поток ввода для функции

ss << a << ' ' << b << ' ' << c << ' ' << d << ' ';

}

// т.к в функции будем считывать поток до конца eof(),

// после последнего набора пробел не

// нужен(соответственно, цикл до k-1)

cin >> a >> b >> c >> d;

ss << a << ' ' << b << ' ' << c << ' ' << d;

vec t = create(ss);

for (int i = 0; i < k; i++) {

cout << t[i].first.first << " " << t[i].first.second << " ";

cout << t[i].second.first << " " << t[i].second.second << endl;

}

return 0;

}

Юнит-тест для этой функции должен сравнить размеры векторов, и если они равны поэлементно сравнить значения:

TEST\_METHOD(TestMethod\_compare\_element) {

stringstream ss;

ss << "-5 0 -2 4 -1 0 0 1 1 0 5 0"; //создаем поток ввода

vec t\_actual = create(ss);

// обратить внимание на разные способы задания значений

// элементам вектора сложной структуры

vec t\_exprcted = { { {-5, 0}, {-2, 4} },

{ make\_pair(-1, 0), {0, 1} },

make\_pair(make\_pair(1, 0), make\_pair(5, 0))

};

// Если размеры векторов различны, то выведется сообщение,

// указанное в третьем параметре AreEqual() – тип строки

// wstring (2 байта на каждый символ) поэтому нужен

// квалификатор «L» перед строковой константой.

Assert::AreEqual(t\_actual.size(), t\_exprcted.size(), L" different size");

// Здесь размеры одинаковые, сверяем координаты

for (unsigned int i = 0; i < t\_actual.size(); i++) {

Assert::AreEqual(t\_actual[i].first.first, t\_exprcted[i].first.first);

Assert::AreEqual(t\_actual[i].first.second, t\_exprcted[i].first.second);

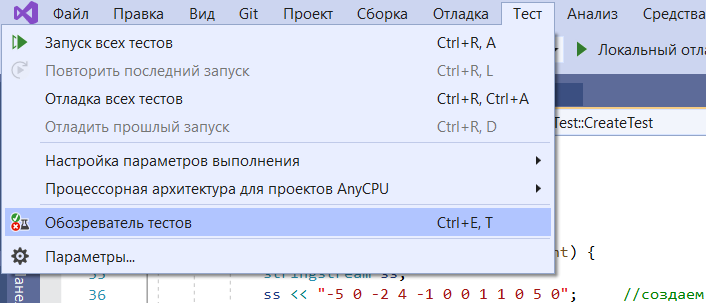
Assert::AreEqual(t\_actual[i].second.first, t\_exprcted[i].second.first);

Assert::AreEqual(t\_actual[i].second.second, t\_exprcted[i].second.second);

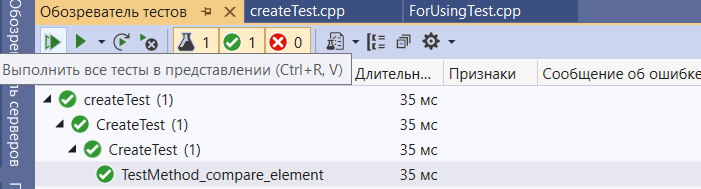
}

}

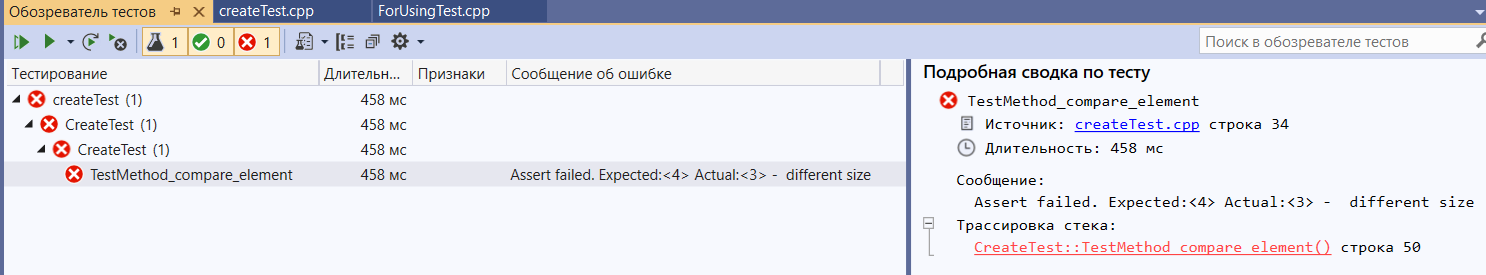
Для запуска тестов на исполнение в Visual Studio удобно использовать окно «Обозреватель тестов», его можно открыть либо из пункта меню «Тест», либо горячими клавишами. Обозреватель тестов может быть открыт в отдельном окне или добавлен в закладки главного окна Visual Studio.



При успешном прохождении теста, он будет отечен зеленой галочкой:



Если добавить еще координаты точек в поток ввода, длина построенного вектора будет отличатся от длины ожидаемого вектора и тест не пройдет, об этом пользователя информирует сообщение, которое мы вставили выше. Раскрыв все треугольные стрелки и выделив тест можно посмотреть подробности:



Сравнивая два последних изображения, видим, что в обозревателе тестов отображается общее количество тестов, количество пройденых тестов и количество не пройденых тестов.

Для контейнера pair<> в С++ определена операция сравнение, попробуем сократить наш тест:

TEST\_METHOD(TestMethod\_compare\_pair) {

stringstream ss;

ss << "-5 0 -2 4 -1 0 0 1 1 0 5 0";

vec t\_actual = create(ss);

vec t\_exprcted = { { {-5, 0}, {-2, 4} },

{ {-1, 0}, {0, 1} },

{ {1, 0}, {5, 0} }

};

Assert::AreEqual(t\_actual.size(), t\_exprcted.size(), L" different size");

for (unsigned int i = 0; i < t\_actual.size(); i++) {

Assert::AreEqual(t\_actual[i].first, t\_exprcted[i].first);

Assert::AreEqual(t\_actual[i].second, t\_exprcted[i].second);

}

}

и сразу же получим ошибку компиляции:

*C2338:*

*Test writer must define specialization of ToString<const Q& q> for your class*

*class std::basic\_string<wchar\_t,struct std::char\_traits<wchar\_t>,*

*class std::allocator<wchar\_t> > \_\_cdecl Microsoft::VisualStudio::CppUnitTestFramework::*

*ToString<struct std::pair< double, double>>(const struct std::pair< double, double> &).*

Эта ошибка возникает по простой причине: в методе AreEqual есть макрос EQUALS\_MESSAGE, который вызывает функцию *ToString()* для вывода сообщений: Expected:<...> Actual:<...>

Шаблон для функции *ToString()* определен только для простых типов (int, double, char, string и длинных/коротких, беззнаковых и т.п), другими словами *ToString()* не умеет выводить пары.

Добавим соответствующий шаблон *ToString()*, который по pair< double, double> построит строку wstring wstr;

namespace Microsoft {

namespace VisualStudio {

namespace CppUnitTestFramework {

template <> static std::wstring ToString(const pair<double, double>& q)

{

wstringstream s;

s << q.first << ' ' << q.second << ' ';

return s.str();

}

}

}

}

Стоит отметить, что использование пространства имен

namespace Microsoft {

namespace VisualStudio {

namespace CppUnitTestFramework {

// ...

}

}

}

Не эквивалентно указанию

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

(в разнице предлагается разобраться в качестве упражнения)

Возникает вопрос: Для контейнера vector<> тоже определена операция сравнения, можно ли еще сократить тест?

TEST\_METHOD(TestMethod\_compare\_vector) {

stringstream ss;

ss << "-5 0 -2 4 -1 0 0 1 1 0 5 0";

vec t\_exprcted = { {{-5, 0}, {-2, 4}}, {{-1, 0}, {0, 1}}, {{1, 0}, {5, 0}} };

Assert::AreEqual(create(ss), t\_exprcted);

}

После определения соответствующего *ToString()*, такой тест проходит, причем, в поток можно вставлять любые разделители, для удобства восприятия структуры данных.

template <> static std::wstring ToString(const vec& q) {

wstringstream s;

for (auto i = q.begin(); i != q.end(); i++) {

s << "{ {" << i->first.first << ", " << i->first.second << "}, {";

s << i->second.first << ", " << i->second.second << "} }, ";

}

return s.str();

}

Попробуем задать числа с дробной частью в поток ввода и в ожидаемый вектор.

TEST\_METHOD(TestMethod\_compare\_vector\_double) {

stringstream ss;

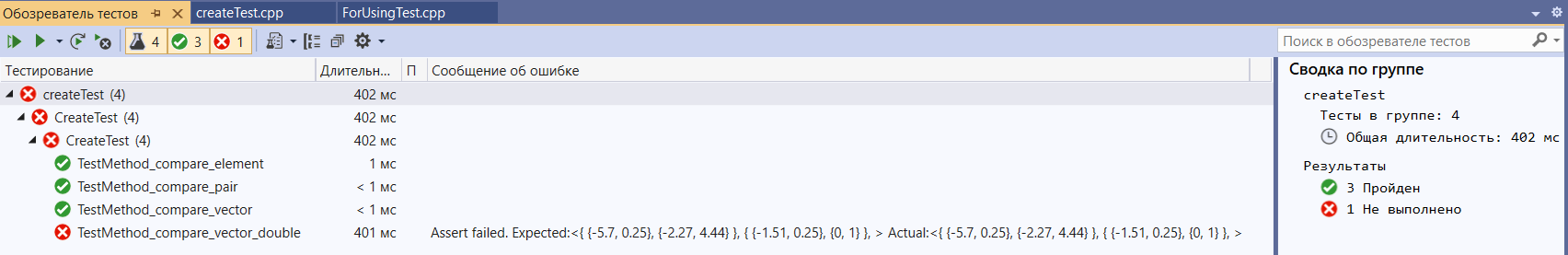
ss << "-5.7 0.25 -2.27 4.44 -1.51 0.25 0.0 1.0";

vec t\_exprcted = { {{-5.7, 0.25}, {-2.27, 4.44}}, {{-1.51, 0.25}, {0.0, 1.0}} };

Assert::AreEqual(create(ss), t\_exprcted);

}

Запуск тестов нам дает такую картину:



Видим, что три теста прошли, а последний нет, но значения в Expected:<...> и Actual:<...> одинаковые. Проблема в том, что вещественные числа хранятся в памяти компьютера с некоторой точностью

**Пример 2.** Напишем функцию, которая по вектору с координатами концов отрезков строит вектор их длин.

typedef vector < double> l\_vec;

l\_vec lenght\_vec(vec a) {

l\_vec rez;

for (auto i = a.begin(); i != a.end(); i++) {

double r = sqrt(pow((\*i).first.first - (\*i).second.first, 2) +

pow((\*i).first.second - (\*i).second.second, 2));

rez.push\_back(r);

}

return rez;

}

Соответственно, *ToString()*:

template <> static std::wstring ToString(const l\_vec& q) {

stringstream s;

s.precision(4); // можно указать сколько знаков выводить в поток

for (auto i = q.begin(); i != q.end(); i++) s << \*i << ' ';

string str = s.str();

wstring wstr(str.begin(), str.end());

return wstr;

}

И тест:

TEST\_METHOD(TestMethod1) {

vec t\_input = { { {-5, 0}, {-2, 4} },

{ {-1, 0}, {0, 1} },

{ {1, 0}, {5, 0} }

};

Assert::AreEqual(lenght\_vec(t\_input), {5, 1.41421, 4});

}

Который выдаст, как и ожидалось:

Assert failed. Expected:<5 1.414 4 > Actual:<5 1.414 4 >

несмотря на то, что координаты точек подобраны так, что длины отрезков считаются хорошо (Пифагоров треугольник, √2 и две точки на одной прямой).

Проблема, естественно, в типе double: сколько знаков после запятой указать в t\_exprcted для √2? В общем случае сравнения чисел плавающей запятой некорректны.

Выход - научить *AreEqual()* сравнивать элементы вектора, с некоторой точностью. Для этого пишем свою операцию сравнения.

bool operator == (l\_vec a, l\_vec b) {

bool ok = true;

if (a.size() == b.size()) {

auto j = b.begin();

for(auto i = a.begin(); i != a.end(); i++)

if (!(ok = abs(\*i - \*(j++)) < 0.00001)) break;

return ok;

}

else return false;

}

Отлично, но если написать тест через поэлементное сравнение векторов:

TEST\_METHOD(TestMethod2) {

vec t\_input = { { {-5, 0}, {-2, 4} },

{ {-1, 0}, {0, 1} },

{ {1, 0}, {5, 0} }

};

vector< double> r\_exprcted = { 5, 1.41421, 4 };

vector< double> r\_actual = lenght\_vec(t\_input);

for (unsigned int i = 0; i < r\_actual.size(); i++) {

Assert::AreEqual(r\_actual[i], r\_exprcted[i]);

}

то для √2 этот тест не пройдет, а переопределять операцию сравнения для стандартных типов нельзя. Выходом может быть запись определенного количества знаков после запятой при вычислении длины отрезка в функции l\_vec lenght\_vec(vec a).

rez.push\_back( ( double) ( (int)(r \* 100000) ) / 100000 ); вместо rez.push\_back(r);

**Что и как тестировать**

Следует убедиться, что во время тестирования задействованы все операторы тестируемой функции, то есть обеспечивайте достаточно большой набор вариантов входных параметров, достаточный для тестирования всех случаев каждого ветвления, входа во все циклы и т.п.

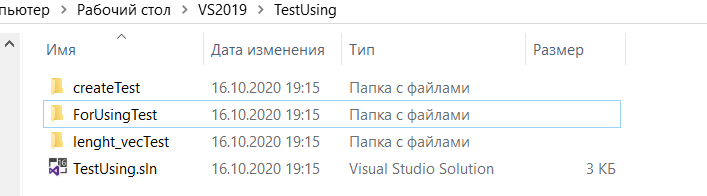
Тестирование циклов (неофициально называемое *«тест 0, 1, 2»*) базируется на том, что если в коде есть цикл, то, чтобы убедиться в его работоспособности, нужно его выполнить 0, 1 и 2 раз. Если он работает корректно во второй итерации, то можно ожидать его корректную работу и для всех последующих итераций (3, 4, 10, 100 и т.д.).

Тестируйте разные типы ввода, чтобы убедиться, что ваш «кусок кода» правильно их обрабатывает.

* + Для целых чисел убедитесь, что вы проверили, как ваша функция обрабатывает 0, отрицательные и положительные значения. При наличии пользовательского ввода вы также должны проверить вариант возникновения переполнения.
  + Для чисел типа с плавающей запятой убедитесь, что вы рассмотрели варианты, как ваша функция обрабатывает значения, которые имеют неточности (значения, которые немного больше/меньше ожидаемых).
  + Для строк убедитесь, что вы рассмотрели вариант, как ваша функция обрабатывает пустую строку, строку с допустимыми значениями, строку с пробелами и строку, содержимым которой являются одни пробелы.

В заключении стоит отметить, что все тесты должны быть независимы, т.е. не стоит в одном TestMethod вызывать все функции. Более того, тесты для разных функций стоит поместить в разные классы.

**Полный код решения примера**

Решение *TestUsing.sln* содержит три проекта

* *ForUsingTest* – проект в котором описаны функции для тестирования. Полное содержимое файла *ForUsingTest.cpp*:

#include <vector>

#include <iterator>

#include <utility>

#include <iostream>

#include <sstream>

using namespace std;

typedef vector < pair<pair< double, double>, pair< double, double> > > vec;

typedef vector < double> l\_vec;

bool operator == (l\_vec a, l\_vec b) {

bool ok = true;

if (a.size() == b.size()) {

auto j = b.begin();

for (auto i = a.begin(); i != a.end(); i++)

if (!(ok = abs(\*i - \*(j++)) < 0.00001)) break;

return ok;

}

else return false;

}

vec create(stringstream& l\_cin) {

vec rez;

while (!l\_cin.eof()) {

double x1, y1, x2, y2;

l\_cin >> x1 >> y1 >> x2 >> y2;

rez.push\_back(make\_pair(make\_pair(x1, y1), make\_pair(x2, y2)));

}

return rez;

}

l\_vec lenght\_vec(vec a) {

l\_vec rez;

for (auto i = a.begin(); i != a.end(); i++) {

double r = sqrt(pow((\*i).first.first - (\*i).second.first, 2) +

pow((\*i).first.second - (\*i).second.second, 2));

rez.push\_back( ( double) ( (int)(r \* 100000) ) /100000 );

//rez.push\_back(r);

}

return rez;

}

int main() {

int k;

cin >> k;

std::stringstream ss;

int a, b, c, d;

for (int i = 0; i < k - 1; i++) {

cin >> a >> b >> c >> d;

ss << a << ' ' << b << ' ' << c << ' ' << d << ' ';

}

cin >> a >> b >> c >> d;

ss << a << ' ' << b << ' ' << c << ' ' << d;

vec t = create(ss);

l\_vec rt = lenght\_vec(t);

for (int i = 0; i < k; i++) {

cout << rt[i] << " == " << t[i].first.first << " "

<< t[i].first.second << " ";

cout << t[i].second.first << " " << t[i].second.second << endl;

}

return 0;

}

* *createTest* – тесты для функции create(); файл *createTest.cpp*:

#include "pch.h"

#include "CppUnitTest.h"

#include "../ForUsingTest/ForUsingTest.cpp"

namespace Microsoft {

namespace VisualStudio {

namespace CppUnitTestFramework {

template <> static std::wstring ToString(const vec& q) {

stringstream s;

for (auto i = q.begin(); i != q.end(); i++) {

s << "{{" << i->first.first << ", " <<

i->first.second << "},{" <<

i->second.first << ", " <<

i->second.second << "}}, ";

}

string str = s.str();

wstring wstr(str.begin(), str.end());

return wstr;

}

template <> static std::wstring ToString(const pair< double, double>& q)

{

stringstream s;

s << q.first << ' ' << q.second << ' ';

string str = s.str();

wstring wstr(str.begin(), str.end());

return wstr;

}

}

}

}

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace CreateTest

{

TEST\_CLASS(CreateTest) {

public:

TEST\_METHOD(TestMethod1) {

stringstream ss;

ss << "-5 0 -2 4 -1 0 0 1 1 0 5 0";

vec t\_actual = create(ss);

vec t\_exprcted = { { { -5, 0 }, { -2, 4 } },

{ make\_pair(-1, 0), {0, 1} },

make\_pair(make\_pair(1, 0), make\_pair(5, 0)) };

Assert::AreEqual(t\_actual.size(), t\_exprcted.size(),

L" different size");

for (unsigned int i = 0; i < t\_actual.size(); i++) {

Assert::AreEqual(t\_actual[i].first.first,

t\_exprcted[i].first.first);

Assert::AreEqual(t\_actual[i].first.second,

t\_exprcted[i].first.second);

Assert::AreEqual(t\_actual[i].second.first,

t\_exprcted[i].second.first);

Assert::AreEqual(t\_actual[i].second.second,

t\_exprcted[i].second.second);

}

}

TEST\_METHOD(TestMethod2) {

stringstream ss;

ss << "-5 0 -2 4 -1 0 0 1 1 0 5 0";

vec t\_actual = create(ss);

vec t\_exprcted = { { {-5, 0}, {-2, 4} },

{ {-1, 0}, {0, 1} },

{ {1, 0}, {5, 0} } };

Assert::AreEqual(t\_actual.size(), t\_exprcted.size(),

L" different size");

for (unsigned int i = 0; i < t\_actual.size(); i++) {

Assert::AreEqual(t\_actual[i].first, t\_exprcted[i].first);

Assert::AreEqual(t\_actual[i].second, t\_exprcted[i].second);

}

}

TEST\_METHOD(TestMethod3) {

stringstream ss;

ss << "-5.7 0.25 -2.27 4.44 -1 0 0 1 1 0 5 0";

vec t\_exprcted = { {{-5.7, 0.25}, {-2.27, 4.44}},

{{-1, 0},{0, 1}}, {{1, 0},{5, 0}} };

Assert::AreEqual(create(ss), t\_exprcted);

}

TEST\_METHOD(TestMethod4) { // тестируем пустой поток и пустой вектор

stringstream ss;

ss.setstate (ios::eofbit); // установить бит eof, именно его

// состояние проверяет функция еоf()

vec t\_exprcted = {};

Assert::AreEqual(create(ss), t\_exprcted);

}

};

}

* *lenght\_vecTest* – тесты для функции *lenght\_vec()* файл *lenght\_vecTest.cpp*:

#include "pch.h"

#include "CppUnitTest.h"

#include "../ForUsingTest/ForUsingTest.cpp"

namespace Microsoft {

namespace VisualStudio {

namespace CppUnitTestFramework {

template <> static std::wstring ToString(const l\_vec& q) {

stringstream s;

s.precision(4);

for (auto i = q.begin(); i != q.end(); i++) s << \*i << ' ';

string str = s.str();

wstring wstr(str.begin(), str.end());

return wstr;

}

}

}

}

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace LenghtvecTest

{

TEST\_CLASS(LenghtvecTest) {

public:

TEST\_METHOD(TestMethod1) {

vec t\_input = { { {-5, 0}, {-2, 4} }, { {-1, 0}, {0, 1} },

{ {1, 0}, {5, 0} } };

Assert::AreEqual(lenght\_vec(t\_input), { 5, 1.41421, 4 });

}

TEST\_METHOD(TestMethod2) {

vec t\_input = { { {-5, 0}, {-2, 4} }, { {-1, 0}, {0, 1} },

{ {1, 0}, {5, 0} } };

vector< double> r\_exprcted = { 5, 1.41421, 4 };

vector< double> r\_actual = lenght\_vec(t\_input);

for (unsigned int i = 0; i < r\_actual.size(); i++)

Assert::AreEqual(r\_actual[i], r\_exprcted[i]);

}

};

}