Министерство цифрового развития, связи и массовых коммуникаций Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Сибирский государственный университет телекоммуникаций и информатики» (СибГУТИ)

## Кафедра прикладной математики и кибернетики

Современные технологии программирования

Лабораторная работа №5

«Разработка и модульное тестирование абстрактного типа данных (ADT) p-ичное число C++»

Выполнил: студент 4 курса

группы ИП-111 Кузьменок Д.В.

Проверил преподаватель: Зайцев Михаил Георгиевич

Новосибирск, 2024 г.

# Цель:

Сформировать практические навыки: реализации абстрактного типа данных с помощью классов С++ и их модульного тестирования.

# Задание:

1. Реализовать абстрактный тип данных «р-ичное число», используя класс, в соответствии с приведенной ниже спецификацией.
2. Протестировать каждую операцию, определенную на типе данных по критерию С2, используя средства модульного тестирования Visual Studio.
3. Если необходимо, предусмотрите возбуждение исключительных ситуаций

# Данные:

Р-ичное число TPNumber - это действительное число (n) со знаком в системе счисления с основанием (b) (в диапазоне 2..16), содержащее целую и дробную части.

Точность представления числа – (c >= 0). Р-ичные числа неизменяемые.

# Операции

Операции могут вызываться только объектом р-ичное число (тип TPNumber), указатель на который в них передаётся по умолчанию. При описании операций этот объект называется this «само число».

# Реализация:

Класс TPNumber представляет собой модель для работы с числами в различных системах счисления (от 2 до 16) с заданной точностью, позволяя выполнять арифметические операции с числами в нестандартных системах счисления. Рассмотрим его реализацию более подробно.

## Поля класса:

* double number: число, которое хранится в объекте, представленное в десятичной системе.
* int base: основание системы счисления (от 2 до 16).
* int precision: количество знаков после запятой при представлении числа.

## Описание методов класса TPNumber Конструкторы:

1. **Конструктор по умолчанию:** TPNumber();

## Входные данные: Нет.

Инициализирует объект с числом 1.0, основанием системы счисления 10, и точностью 1.

**Выходные данные:** Объект класса TPNumber с параметрами по умолчанию.

## Конструктор с параметрами (число, основание, точность):

TPNumber(double a, int b, int c);

## Входные данные:

* + a: Число в десятичной системе (тип double).
  + b: Основание системы счисления (тип int, должно быть в диапазоне от 2 до 16).
  + c: Точность (количество знаков после запятой, тип int, неотрицательное).

Инициализирует объект числом a, основанием b, и точностью c. Если входные данные не соответствуют ограничениям, выбрасывается исключение invalid\_argument.

**Выходные данные:** Объект класса TPNumber с заданными значениями или значениями по умолчанию при некорректных параметрах.

1. **Конструктор с параметрами (число, основание, точность в виде строк):** TPNumber(string a, string b, string c);

## Входные данные:

* + a: Число в виде строки (тип string).
  + b: Основание системы счисления в виде строки (тип string).
  + c: Точность в виде строки (тип string).

Преобразует входные строки в числовые значения, проверяет их на корректность (основание от 2 до 16, точность неотрицательна). Если проверка успешна, инициализирует объект этими значениями, иначе — выбрасывается исключение invalid\_argument.

**Выходные данные:** Объект класса TPNumber.

## Арифметические методы:

1. **Метод сложения:**

TPNumber add(TPNumber right); TPNumber operator+(TPNumber right);

## Входные данные:

* + right: Объект класса TPNumber, который будет добавлен к текущему объекту.

Выполняет сложение двух объектов, если у них одинаковое основание системы счисления и точность. Если основание или точность различаются, возвращает объект с нулевым числом, основанием 10, и точностью 0.

**Выходные данные:** Новый объект класса TPNumber, представляющий сумму двух чисел или нулевое значение при различии оснований или точностей.

## Метод вычитания:

TPNumber subtr(TPNumber right); TPNumber operator-(TPNumber right);

## Входные данные:

* + right: Объект класса TPNumber, который будет вычтен из текущего объекта.

Вычитает right из текущего объекта, если у них одинаковое основание и точность. В противном случае возвращает объект с нулевыми значениями.

**Выходные данные:** Новый объект класса TPNumber с результатом вычитания или нулевыми значениями.

## Метод умножения:

TPNumber mult(TPNumber right); TPNumber operator\*(TPNumber right);

## Входные данные:

* + right: Объект класса TPNumber, который будет умножен на текущий объект.

Выполняет умножение двух объектов с одинаковым основанием и точностью. Если они различаются, возвращает объект с нулевыми значениями.

**Выходные данные:** Объект класса TPNumber, содержащий результат умножения или нулевые значения.

## Метод деления:

TPNumber div(TPNumber right); TPNumber operator/(TPNumber right);

## Входные данные:

* + right: Объект класса TPNumber, на который будет делиться текущее число.

Делит текущий объект на right, если у них одинаковое основание и точность, и если значение right не равно нулю. При делении на ноль выбрасывает исключение.

**Выходные данные:** Объект класса TPNumber, содержащий результат деления, или выбрасывается исключение при делении на ноль.

## Прочие методы:

1. **Метод возведения в квадрат:** TPNumber square();

## Входные данные: Нет.

Возводит текущее число в квадрат.

**Выходные данные:** Объект класса TPNumber с результатом возведения в квадрат.

1. **Метод обратного числа:** TPNumber inverse();

## Входные данные: Нет.

Возвращает обратное значение текущего числа (единица, деленная на число).

**Выходные данные:** Объект класса TPNumber, представляющий обратное число.

## Методы получения и изменения числа, основания и точности:

double getNumber();

string getLeftPartString();

string getRightPartString();

string getNumberString(); int getBase();

string getBaseString(); int getPrecision();

string getPrecisionString(); void setBase(int newBase); void setBase(string stringBase);

void setPrecision(int newPrecision); void setPrecision(string stringPrecision);

## Входные данные:

* + Методы setBase, setPrecision: принимают новое основание или точность как int или string.
  + Методы getNumber, getBase, getPrecision: не принимают входных данных.

## Процесс:

* + - getNumber возвращает текущее значение числа.
    - getNumberString возвращает строковое представление числа в текущей системе счисления.
    - getBase возвращает текущее основание системы счисления.
    - getBaseString возвращает строковое представление текущего основания.
    - getPrecision возвращает текущую точность.
    - getPrecisionString возвращает строковое представление точности.
    - setBase и setPrecision изменяют текущее основание и точность, если параметры валидны (основание между 2 и 16, точность >= 0).

**Выходные данные:** В зависимости от метода — строка, число, или модификация параметров объекта.

**Метод копирования:** TPNumber copy();

## Входные данные: Нет.

Создает копию текущего объекта.

**Выходные данные:** Новый объект TPNumber с теми же значениями.

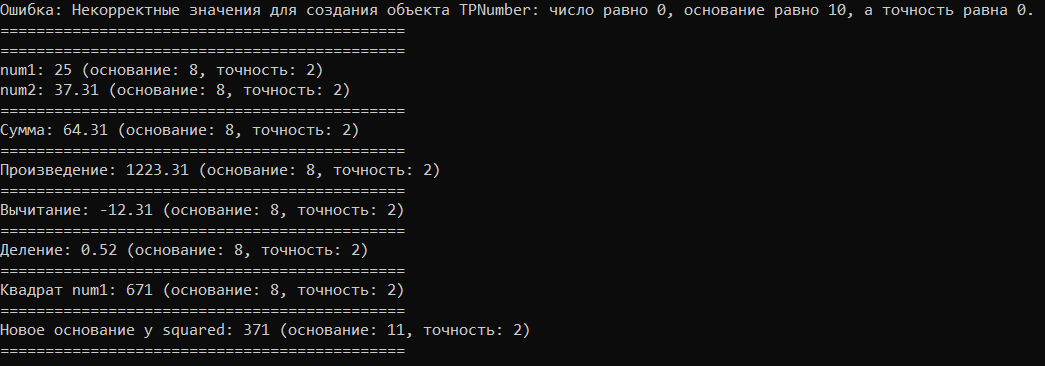
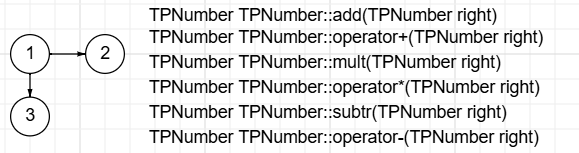
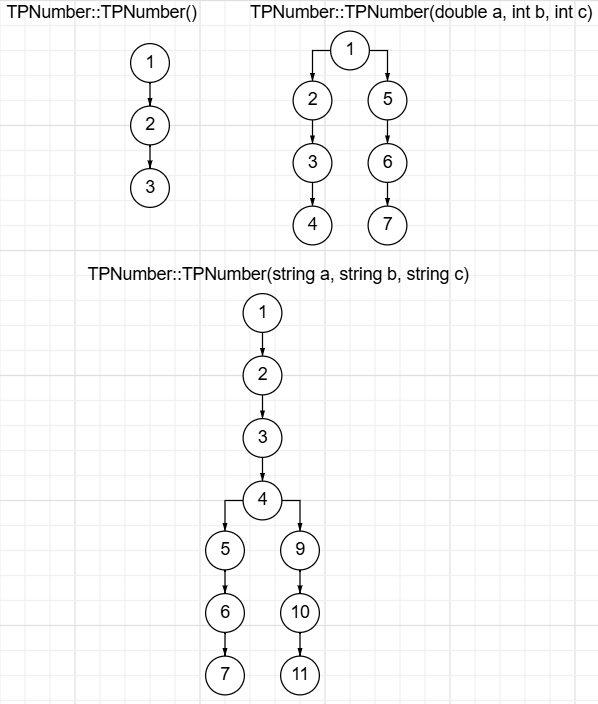
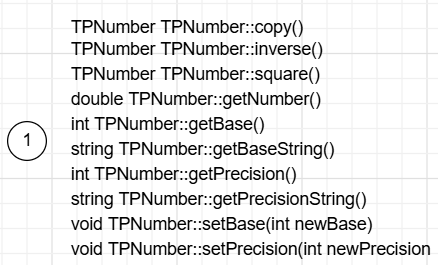
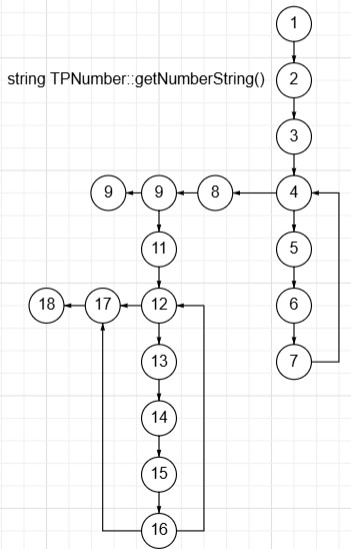
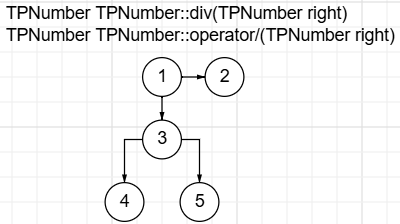


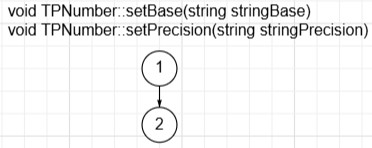
Рис. 1 – Результат проверки работоспособности программы.

По готовым функциям, были построены управляющие графы программы:









# Описание тестовых данных

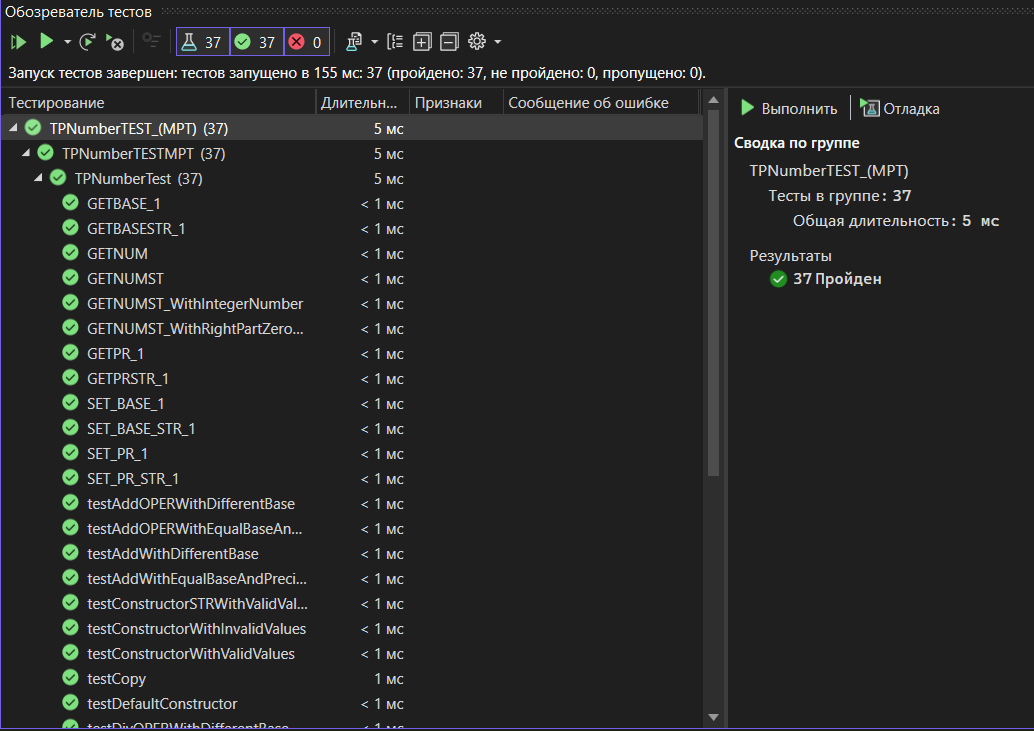


Рис. 2 – Результат выполнения модульных тестов.

**Описание тестовых наборов:**

**testDefaultConstructor**: проверяет, что конструктор по умолчанию устанавливает значения числа, базы и точности равными 1.0, 10 и 1 соответственно.

**testConstructorWithValidValues**: проверяет, что конструктор с заданными значениями корректно инициализирует число, базу и точность.

**testConstructorWithInvalidValues**: проверяет, что конструктор с недопустимыми значениями (база больше 16 и отрицательная точность) устанавливает значения числа, базы и точности по умолчанию.

**testConstructorSTRWithValidValues**: проверяет, что конструктор, принимающий строки, корректно инициализирует число, базу и точность.

**testCopy**: проверяет, что метод копирования создает корректную копию объекта TPNumber.

**testAddWithEqualBaseAndPrecision**: проверяет, что сложение двух чисел с одинаковыми базами и точностями возвращает правильный результат.

**testAddWithDifferentBase**: проверяет, что сложение двух чисел с разными базами возвращает значение по умолчанию.

**testMultWithEqualBaseAndPrecision**: проверяет, что умножение двух чисел с одинаковыми базами и точностями возвращает правильный результат.

**testMultWithDifferentBase**: проверяет, что умножение двух чисел с разными базами возвращает значение по умолчанию.

**testSubtrWithEqualBaseAndPrecision**: проверяет, что вычитание двух чисел с одинаковыми базами и точностями возвращает правильный результат.

**testSubtrWithDifferentBase**: проверяет, что вычитание двух чисел с разными базами возвращает значение по умолчанию.

**testDivWithEqualBaseAndPrecision**: проверяет, что деление двух чисел с одинаковыми базами и точностями возвращает правильный результат.

**testDivWithZeroDiv**: проверяет, что деление на ноль выбрасывает исключение std::logic\_error.

**testDivWithDifferentBase**: проверяет, что деление двух чисел с разными базами возвращает значение по умолчанию.

**testAddOPERWithEqualBaseAndPrecision**: проверяет, что оператор сложения (+) для двух чисел с одинаковыми базами и точностями возвращает правильный результат.

**testAddOPERWithDifferentBase**: проверяет, что оператор сложения для двух чисел с разными базами возвращает значение по умолчанию.

**testMultOPERWithEqualBaseAndPrecision**: проверяет, что оператор умножения (\*) для двух чисел с одинаковыми базами и точностями возвращает правильный результат.

**testMultOPERWithDifferentBase**: проверяет, что оператор умножения для двух чисел с разными базами возвращает значение по умолчанию.

**testSubtrOPERWithEqualBaseAndPrecision**: проверяет, что оператор вычитания (-) для двух чисел с одинаковыми базами и точностями возвращает правильный результат.

**testSubtOPERrWithDifferentBase**: проверяет, что оператор вычитания для двух чисел с разными базами возвращает значение по умолчанию.

**testDivOPERWithEqualBaseAndPrecision**: проверяет, что оператор деления (/) для двух чисел с одинаковыми базами и точностями возвращает правильный результат.

**testDivOPERWithZeroDiv**: проверяет, что оператор деления на ноль выбрасывает исключение std::logic\_error.

**testDivOPERWithDifferentBase**: проверяет, что оператор деления для двух чисел с разными базами возвращает значение по умолчанию.

**testInverse**: проверяет, что метод инверсии (inverse()) возвращает правильный результат для заданного числа.

**testSquare**: проверяет, что метод возведения в квадрат (square()) возвращает правильный результат для заданного числа.

**GETNUM**: проверяет, что метод получения числа (getNumber()) возвращает правильное значение.

**GETNUMST**: проверяет, что метод получения строки числа (getNumberString()) возвращает корректное строковое представление числа в заданной базе.

**GETNUMST\_WithRightPartZeroBreak**: проверяет, что метод получения строки числа корректно обрабатывает число с дробной частью и возвращает правильное представление.

**GETNUMST\_WithIntegerNumber**: проверяет, что метод получения строки числа корректно обрабатывает целое число и возвращает правильное представление.

**GETBASE\_1**: проверяет, что метод получения базы (getBase()) возвращает правильное значение.

**GETBASESTR\_1**: проверяет, что метод получения строки базы (getBaseString()) возвращает правильное строковое представление базы.

**GETPR\_1**: проверяет, что метод получения точности (getPrecision()) возвращает правильное значение.

**GETPRSTR\_1**: проверяет, что метод получения строки точности (getPrecisionString()) возвращает правильное строковое представление точности.

**SET\_BASE\_1:** проверяет, правильно ли метод setBase(int base) устанавливает основание числа.

**SET\_BASE\_STR\_1:** проверяет, правильно ли метод setBase(std::string base) устанавливает основание числа из строкового представления.

**SET\_PR\_1:** проверяет, правильно ли метод setPrecision(int precision) устанавливает точность числа.

**SET\_PR\_STR\_1:** проверяет, правильно ли метод setPrecision(std::string precision) устанавливает точность числа из строкового представления.

# Вывод:

В ходе выполнения лабораторной работы были успешно освоены практические навыки объектно-ориентированного программирования на языке C++. В частности, были получены опыт разработки функций классов, написания модульных тестов с использованием и применения инструментов автоматизации Visual Studio для проведения модульного тестирования.

# Листинг программы:

## TPNumber.h:

#pragma once

#include <string>

**using** **namespace** std;

**class** **TPNumber**

{

**private:**

**double** number;

**int** base, precision;

**public:**

TPNumber();

~TPNumber();

TPNumber(**double**, **int**, **int**);

TPNumber(string, string, string);

TPNumber **copy**();

TPNumber **inverse**();

TPNumber **square**();

TPNumber **add**(TPNumber);

TPNumber **operator**+(TPNumber);

TPNumber **subtr**(TPNumber);

TPNumber **operator**-(TPNumber);

TPNumber **mult**(TPNumber);

TPNumber **operator**\*(TPNumber);

TPNumber **div**(TPNumber);

TPNumber **operator**/(TPNumber);

**double** **getNumber**();

string **getLeftPartString**();

string **getRightPartString**();

string **getNumberString**();

**int** **getBase**();

string **getBaseString**();

**int** **getPrecision**();

string **getPrecisionString**();

**void** **setBase**(**int**);

**void** **setBase**(string);

**void** **setPrecision**(**int**);

**void** **setPrecision**(string);

**int** **equal**(TPNumber, TPNumber);

};

## TPNumber.cpp:

#include "TPNumber.h"

#include <cmath>

#include <stdexcept>

**using** **namespace** std;

TPNumber::TPNumber() {

**this**->number = **1.0**;

**this**->base = **10**;

**this**->precision = **1**;

}

TPNumber::TPNumber(**double** a, **int** b, **int** c) {

**if** (a != **0** && b >= **2** && b <= **16** && c >= **0**) {

**this**->number = a;

**this**->base = b;

**this**->precision = c;

}

**else** {

**throw** std::invalid\_argument("Некорректные значения для создания объекта TPNumber: число равно 0, основание равно 10, а точность равна 0.");

}

}

TPNumber::TPNumber(string a, string b, string c) {

**double** numberTemp = stod(a);

**int** baseTemp = stoi(b);

**int** precisionTemp = stoi(c);

**if** (numberTemp != **0** && baseTemp >= **2** && baseTemp <= **16** && precisionTemp >= **0**) {

**this**->number = numberTemp;

**this**->base = baseTemp;

**this**->precision = precisionTemp;

}

**else** {

**throw** std::invalid\_argument("Некорректные значения для создания объекта TPNumber: число равно 0, основание равно 10, а точность равна 0.");

}

}

TPNumber TPNumber::copy() {

**return** { number, base, precision };

}

TPNumber TPNumber::add(TPNumber right) {

**if** (equal(\***this**, right)) {

**return** { **this**->number + right.number, **this**->base, **this**->precision };

}

**else** **return** { **0.0**, **10**, **0** };

}

TPNumber TPNumber::**operator**+(TPNumber right) {

**if** (equal(\***this**, right)) {

**return** { **this**->number + right.number, **this**->base, **this**->precision };

}

**else** **return** { **0.0**, **10**, **0** };

}

TPNumber TPNumber::mult(TPNumber right) {

**if** (equal(\***this**, right)) {

**return** { **this**->number \* right.number, **this**->base, **this**->precision };

}

**else** **return** { **0.0**, **10**, **0** };

}

TPNumber TPNumber::**operator**\*(TPNumber right) {

**if** (equal(\***this**, right)) {

**return** { **this**->number \* right.number, **this**->base, **this**->precision };

}

**else** **return** { **0.0**, **10**, **0** };

}

TPNumber TPNumber::subtr(TPNumber right) {

**if** (equal(\***this**, right)) {

**return** { **this**->number - right.number, **this**->base, **this**->precision };

}

**else** **return** { **0.0**, **10**, **0** };

}

TPNumber TPNumber::**operator**-(TPNumber right) {

**if** (equal(\***this**, right)) {

**return** { **this**->number - right.number, **this**->base, **this**->precision };

}

**else** **return** { **0.0**, **10**, **0** };

}

TPNumber TPNumber::div(TPNumber right) {

**if** (right.number == **0.0**)

**throw** std::logic\_error("Division by zero");

**else** **if** (equal(\***this**, right)) {

**return** { **this**->number / right.number, **this**->base, **this**->precision };

}

**else** **return** { **0.0**, **10**, **0** };

}

TPNumber TPNumber::**operator**/(TPNumber right) {

**if** (right.number == **0.0**)

**throw** std::logic\_error("Division by zero");

**else** **if** (equal(\***this**, right)) {

**return** { **this**->number / right.number, **this**->base, **this**->precision };

}

**else** **return** { **0.0**, **10**, **0** };

}

TPNumber TPNumber::inverse() {

**return** { **1** / **this**->number, **this**->base, **this**->precision };

}

TPNumber TPNumber::square() {

**return** { **this**->number \* **this**->number, **this**->base, **this**->precision };

}

**double** TPNumber::getNumber() {

**return** **this**->number;

}

string TPNumber::getLeftPartString() {

**double** tempNumber = fabs(**this**->number);

**int** leftPart = (**int**)floor(tempNumber); // Целая часть числа

string result;

**while** (leftPart > **0**) {

**int** ost = leftPart % **this**->base;

result.insert(**0**, **1**, symbols[ost]);

leftPart /= **this**->base;

}

**if** (result.empty()) result = "0";

**return** result;

}

string TPNumber::getRightPartString() {

**double** tempNumber = fabs(**this**->number);

**double** rightPart = tempNumber - floor(tempNumber); // Дробная часть числа

string result;

**if** (rightPart == **0.0**) **return** result;

**else** {

result.append(**1**, '.');

**for** (**int** i = **1**; i <= precision; i++) {

rightPart \*= base;

result.append(**1**, symbols[(**int**)floor(rightPart)]);

rightPart -= floor(rightPart);

**if** (rightPart == **0.0**) **break**;

}

}

**return** result;

}

string TPNumber::getNumberString() {

string result = getLeftPartString();

string rightPart = getRightPartString();

**if** (!rightPart.empty()) {

result.append(rightPart);

}

**if** (**this**->number < **0**) result = result = "-" + result;

**return** result;

}

**int** TPNumber::getBase() {

**return** **this**->base;

}

string TPNumber::getBaseString() {

**return** to\_string(**this**->base);

}

**int** TPNumber::getPrecision() {

**return** **this**->precision;

}

string TPNumber::getPrecisionString() {

**return** to\_string(**this**->precision);

}

**void** TPNumber::setBase(**int** newBase) {

**if** (newBase >= **2** && newBase <= **16**) **this**->base = newBase;

}

**void** TPNumber::setBase(string stringBase) {

**int** newBase = stoi(stringBase);

**if** (newBase >= **2** && newBase <= **16**) **this**->base = newBase;

}

**void** TPNumber::setPrecision(**int** newPrecision) {

**if** (newPrecision >= **0**) **this**->precision = newPrecision;

}

**void** TPNumber::setPrecision(string stringPrecision) {

**int** newPrecision = stoi(stringPrecision);

**if** (newPrecision >= **0**) **this**->precision = newPrecision;

}

**int** TPNumber::equal(TPNumber num1, TPNumber num2) {

**if** (num1.base == num2.base && num1.precision == num2.precision) {

**return** true;

}

**return** false;

}

TPNumber::~TPNumber() {}

## main.cpp:

#include "TPNumber.h"

#include <iostream>

#include <Windows.h>

**using** **namespace** std;

**int** **main**() {

SetConsoleCP(**1251**);

SetConsoleOutputCP(**1251**);

TPNumber num1(**21**, **8**, **2**); // Число 21 в восьмиричной системе с точностью 2 знака

TPNumber num2("31.4", "8", "2"); // Число 31.4 в восьмиричной системе с точностью 2 знака

try {

TPNumber num3(**0**, **10**, **2**);

}

**catch** (**const** std::invalid\_argument& e) {

std::cerr << "Ошибка: " << e.what() << std::endl;

cout << "=============================================**\n**";

}

cout << "=============================================**\n**";

cout << "num1: " << num1.getNumberString() << " (основание: " << num1.getBaseString() << ", точность: " << num1.getPrecisionString() << ")" << endl;

cout << "num2: " << num2.getNumberString() << " (основание: " << num2.getBaseString() << ", точность: " << num2.getPrecisionString() << ")" << endl;

cout << "=============================================**\n**";

TPNumber sum = num1 + num2;

cout << "Сумма: " << sum.getNumberString() << " (основание: " << sum.getBaseString() << ", точность: " << sum.getPrecisionString() << ")" << endl;

cout << "=============================================**\n**";

TPNumber product = num1 \* num2;

cout << "Произведение: " << product.getNumberString() << " (основание: " << product.getBaseString() << ", точность: " << product.getPrecisionString() << ")" << endl;

cout << "=============================================**\n**";

TPNumber difference = num1 - num2;

cout << "Вычитание: " << difference.getNumberString() << " (основание: " << difference.getBaseString() << ", точность: " << difference.getPrecisionString() << ")" << endl;

cout << "=============================================**\n**";

TPNumber quotient;

try {

quotient = num1 / num2;

cout << "Деление: " << quotient.getNumberString() << " (основание: " << quotient.getBaseString() << ", точность: " << quotient.getPrecisionString() << ")" << endl;

cout << "=============================================**\n**";

}

**catch** (**const** logic\_error& e) {

cerr << "Error: " << e.what() << endl;

cout << "=============================================**\n**";

}

TPNumber squared = num1.square();

cout << "Квадрат num1: " << squared.getNumberString() << " (основание: " << squared.getBaseString() << ", точность: " << squared.getPrecisionString() << ")" << endl;

cout << "=============================================**\n**";

squared.setBase(**11**);

cout << "Новое основание у squared: " << squared.getNumberString() << " (основание: " << squared.getBaseString() << ", точность: " << squared.getPrecisionString() << ")" << endl;

cout << "=============================================**\n**";

**return** **0**;

}

## TPNumberTEST.cpp:

#include "pch.h"

#include "CppUnitTest.h"

#include "../PNumber\_(MPT5)/TPNumber.h"

**using** **namespace** Microsoft::VisualStudio::CppUnitTestFramework;

**namespace** TPNumberTESTMPT

{

TEST\_CLASS(TPNumberTest)

{

**public:**

TEST\_METHOD(testDefaultConstructor)

{

// Arrange

TPNumber number;

**double** expectedNum = **1.0**;

**int** expectedBase = **10**;

**int** expectedPrecision = **1**;

// Act

**double** num = number.getNumber();

**int** base = number.getBase();

**int** precision = number.getPrecision();

// Assert

Assert::AreEqual(expectedNum, num);

Assert::AreEqual(expectedBase, base);

Assert::AreEqual(expectedPrecision, precision);

}

TEST\_METHOD(testConstructorWithValidValues)

{

// Arrange

TPNumber number(**25.5**, **10**, **3**);

**double** expectedNum = **25.5**;

**int** expectedBase = **10**;

**int** expectedPrecision = **3**;

// Act

**double** num = number.getNumber();

**int** base = number.getBase();

**int** precision = number.getPrecision();

// Assert

Assert::AreEqual(expectedNum, num);

Assert::AreEqual(expectedBase, base);

Assert::AreEqual(expectedPrecision, precision);

}

TEST\_METHOD(testConstructorWithInvalidValues)

{

// Arrange

TPNumber number(**25.5**, **20**, -**1**); // Неверная база и точность

**double** expectedNum = **0.0**;

**int** expectedBase = **10**;

**int** expectedPrecision = **0**;

// Act

**double** num = number.getNumber();

**int** base = number.getBase();

**int** precision = number.getPrecision();

// Assert

Assert::AreEqual(expectedNum, num);

Assert::AreEqual(expectedBase, base);

Assert::AreEqual(expectedPrecision, precision);

}

TEST\_METHOD(testConstructorSTRWithValidValues)

{

// Arrange

TPNumber number("25.5", "10", "3");

**double** expectedNum = **25.5**;

**int** expectedBase = **10**;

**int** expectedPrecision = **3**;

// Act

**double** num = number.getNumber();

**int** base = number.getBase();

**int** precision = number.getPrecision();

// Assert

Assert::AreEqual(expectedNum, num);

Assert::AreEqual(expectedBase, base);

Assert::AreEqual(expectedPrecision, precision);

}

TEST\_METHOD(testCopy)

{

// Arrange

TPNumber number(**4.0**, **10**, **2**);

TPNumber expectedCopy = number;

// Act

TPNumber copy = number.copy();

// Assert

Assert::AreEqual(expectedCopy.getNumber(), copy.getNumber());

Assert::AreEqual(expectedCopy.getBase(), copy.getBase());

Assert::AreEqual(expectedCopy.getPrecision(), copy.getPrecision());

}

TEST\_METHOD(testAddWithEqualBaseAndPrecision)

{

// Arrange

TPNumber num1(**10.5**, **10**, **2**);

TPNumber **num2**(**15.5**, **10**, **2**);

**double** expectedNum = **26.0**;

**int** expectedBase = **10**;

**int** expectedPrecision = **2**;

// Act

TPNumber result = num1.add(num2);

// Assert

Assert::AreEqual(expectedNum, result.getNumber());

Assert::AreEqual(expectedBase, result.getBase());

Assert::AreEqual(expectedPrecision, result.getPrecision());

}

TEST\_METHOD(testAddWithDifferentBase)

{

// Arrange

TPNumber num1(**10.5**, **10**, **2**);

TPNumber **num2**(**15.5**, **2**, **2**);

**double** expectedNum = **0.0**;

**int** expectedBase = **10**;

**int** expectedPrecision = **0**;

// Act

TPNumber result = num1.add(num2);

// Assert

Assert::AreEqual(expectedNum, result.getNumber());

Assert::AreEqual(expectedBase, result.getBase());

Assert::AreEqual(expectedPrecision, result.getPrecision());

}

TEST\_METHOD(testMultWithEqualBaseAndPrecision)

{

// Arrange

TPNumber num1(**5**, **2**, **2**);

TPNumber **num2**(**5**, **2**, **2**);

**double** expectedNum = **25.0**;

**int** expectedBase = **2**;

**int** expectedPrecision = **2**;

// Act

TPNumber result = num1.mult(num2);

// Assert

Assert::AreEqual(expectedNum, result.getNumber());

Assert::AreEqual(expectedBase, result.getBase());

Assert::AreEqual(expectedPrecision, result.getPrecision());

}

TEST\_METHOD(testMultWithDifferentBase)

{

// Arrange

TPNumber num1(**10.5**, **10**, **2**);

TPNumber **num2**(**15.5**, **2**, **2**);

**double** expectedNum = **0.0**;

**int** expectedBase = **10**;

**int** expectedPrecision = **0**;

// Act

TPNumber result = num1.mult(num2);

// Assert

Assert::AreEqual(expectedNum, result.getNumber());

Assert::AreEqual(expectedBase, result.getBase());

Assert::AreEqual(expectedPrecision, result.getPrecision());

}

TEST\_METHOD(testSubtrWithEqualBaseAndPrecision)

{

// Arrange

TPNumber num1(**10.5**, **10**, **2**);

TPNumber **num2**(**15.5**, **10**, **2**);

**double** expectedNum = -**5.0**;

**int** expectedBase = **10**;

**int** expectedPrecision = **2**;

// Act

TPNumber result = num1.subtr(num2);

// Assert

Assert::AreEqual(expectedNum, result.getNumber());

Assert::AreEqual(expectedBase, result.getBase());

Assert::AreEqual(expectedPrecision, result.getPrecision());

}

TEST\_METHOD(testSubtrWithDifferentBase)

{

// Arrange

TPNumber num1(**10.5**, **10**, **2**);

TPNumber **num2**(**15.5**, **2**, **2**);

**double** expectedNum = **0.0**;

**int** expectedBase = **10**;

**int** expectedPrecision = **0**;

// Act

TPNumber result = num1.subtr(num2);

// Assert

Assert::AreEqual(expectedNum, result.getNumber());

Assert::AreEqual(expectedBase, result.getBase());

Assert::AreEqual(expectedPrecision, result.getPrecision());

}

TEST\_METHOD(testDivWithEqualBaseAndPrecision)

{

// Arrange

TPNumber num1(**5**, **10**, **2**);

TPNumber **num2**(**5**, **10**, **2**);

**double** expectedNum = **1.0**;

**int** expectedBase = **10**;

**int** expectedPrecision = **2**;

// Act

TPNumber result = num1.div(num2);

// Assert

Assert::AreEqual(expectedNum, result.getNumber());

Assert::AreEqual(expectedBase, result.getBase());

Assert::AreEqual(expectedPrecision, result.getPrecision());

}

TEST\_METHOD(testDivWithZeroDiv)

{

// Arrange

TPNumber numerator(**20.0**, **10**, **2**);

TPNumber **denominator**(**0.0**, **10**, **2**);

// Act & Assert

Assert::ExpectException<std::logic\_error>([&]() { numerator.div(denominator); });

}

TEST\_METHOD(testDivWithDifferentBase)

{

// Arrange

TPNumber num1(**10.5**, **10**, **2**);

TPNumber **num2**(**15.5**, **2**, **2**);

**double** expectedNum = **0.0**;

**int** expectedBase = **10**;

**int** expectedPrecision = **0**;

// Act

TPNumber result = num1.div(num2);

// Assert

Assert::AreEqual(expectedNum, result.getNumber());

Assert::AreEqual(expectedBase, result.getBase());

Assert::AreEqual(expectedPrecision, result.getPrecision());

}

TEST\_METHOD(testAddOPERWithEqualBaseAndPrecision)

{

// Arrange

TPNumber num1(**10.5**, **10**, **2**);

TPNumber **num2**(**15.5**, **10**, **2**);

**double** expectedNum = **26.0**;

**int** expectedBase = **10**;

**int** expectedPrecision = **2**;

// Act

TPNumber result = num1 + num2;

// Assert

Assert::AreEqual(expectedNum, result.getNumber());

Assert::AreEqual(expectedBase, result.getBase());

Assert::AreEqual(expectedPrecision, result.getPrecision());

}

TEST\_METHOD(testAddOPERWithDifferentBase)

{

// Arrange

TPNumber num1(**10.5**, **10**, **2**);

TPNumber **num2**(**15.5**, **2**, **2**);

**double** expectedNum = **0.0**;

**int** expectedBase = **10**;

**int** expectedPrecision = **0**;

// Act

TPNumber result = num1 + num2;

// Assert

Assert::AreEqual(expectedNum, result.getNumber());

Assert::AreEqual(expectedBase, result.getBase());

Assert::AreEqual(expectedPrecision, result.getPrecision());

}

TEST\_METHOD(testMultOPERWithEqualBaseAndPrecision)

{

// Arrange

TPNumber num1(**5**, **10**, **2**);

TPNumber **num2**(**5**, **10**, **2**);

**double** expectedNum = **25.0**;

**int** expectedBase = **10**;

**int** expectedPrecision = **2**;

// Act

TPNumber result = num1 \* num2;

// Assert

Assert::AreEqual(expectedNum, result.getNumber());

Assert::AreEqual(expectedBase, result.getBase());

Assert::AreEqual(expectedPrecision, result.getPrecision());

}

TEST\_METHOD(testMultOPERWithDifferentBase)

{

// Arrange

TPNumber num1(**10.5**, **10**, **2**);

TPNumber **num2**(**15.5**, **2**, **2**);

**double** expectedNum = **0.0**;

**int** expectedBase = **10**;

**int** expectedPrecision = **0**;

// Act

TPNumber result = num1 \* num2;

// Assert

Assert::AreEqual(expectedNum, result.getNumber());

Assert::AreEqual(expectedBase, result.getBase());

Assert::AreEqual(expectedPrecision, result.getPrecision());

}

TEST\_METHOD(testSubtrOPERWithEqualBaseAndPrecision)

{

// Arrange

TPNumber num1(**10.5**, **10**, **2**);

TPNumber **num2**(**15.5**, **10**, **2**);

**double** expectedNum = -**5.0**;

**int** expectedBase = **10**;

**int** expectedPrecision = **2**;

// Act

TPNumber result = num1 - num2;

// Assert

Assert::AreEqual(expectedNum, result.getNumber());

Assert::AreEqual(expectedBase, result.getBase());

Assert::AreEqual(expectedPrecision, result.getPrecision());

}

TEST\_METHOD(testSubtOPERrWithDifferentBase)

{

// Arrange

TPNumber num1(**10.5**, **10**, **2**);

TPNumber **num2**(**15.5**, **2**, **2**);

**double** expectedNum = **0.0**;

**int** expectedBase = **10**;

**int** expectedPrecision = **0**;

// Act

TPNumber result = num1 - num2;

// Assert

Assert::AreEqual(expectedNum, result.getNumber());

Assert::AreEqual(expectedBase, result.getBase());

Assert::AreEqual(expectedPrecision, result.getPrecision());

}

TEST\_METHOD(testDivOPERWithEqualBaseAndPrecision)

{

// Arrange

TPNumber num1(**5**, **10**, **2**);

TPNumber **num2**(**5**, **10**, **2**);

**double** expectedNum = **1.0**;

**int** expectedBase = **10**;

**int** expectedPrecision = **2**;

// Act

TPNumber result = num1 / num2;

// Assert

Assert::AreEqual(expectedNum, result.getNumber());

Assert::AreEqual(expectedBase, result.getBase());

Assert::AreEqual(expectedPrecision, result.getPrecision());

}

TEST\_METHOD(testDivOPERWithZeroDiv)

{

// Arrange

TPNumber numerator(**20.0**, **10**, **2**);

TPNumber **denominator**(**0.0**, **10**, **2**);

// Act & Assert

Assert::ExpectException<std::logic\_error>([&]() { numerator / denominator; });

}

TEST\_METHOD(testDivOPERWithDifferentBase)

{

// Arrange

TPNumber num1(**10.5**, **10**, **2**);

TPNumber **num2**(**15.5**, **2**, **2**);

**double** expectedNum = **0.0**;

**int** expectedBase = **10**;

**int** expectedPrecision = **0**;

// Act

TPNumber result = num1 / num2;

// Assert

Assert::AreEqual(expectedNum, result.getNumber());

Assert::AreEqual(expectedBase, result.getBase());

Assert::AreEqual(expectedPrecision, result.getPrecision());

}

TEST\_METHOD(testInverse)

{

// Arrange

TPNumber number(**4.0**, **10**, **2**);

**double** expectedNum = **0.25**;

**int** expectedBase = **10**;

**int** expectedPrecision = **2**;

// Act

TPNumber result = number.inverse();

// Assert

Assert::AreEqual(expectedNum, result.getNumber());

Assert::AreEqual(expectedBase, result.getBase());

Assert::AreEqual(expectedPrecision, result.getPrecision());

}

TEST\_METHOD(testSquare)

{

// Arrange

TPNumber number(**4.0**, **10**, **2**);

**double** expectedNum = **16.0**;

**int** expectedBase = **10**;

**int** expectedPrecision = **2**;

// Act

TPNumber result = number.square();

// Assert

Assert::AreEqual(expectedNum, result.getNumber());

Assert::AreEqual(expectedBase, result.getBase());

Assert::AreEqual(expectedPrecision, result.getPrecision());

}

TEST\_METHOD(GETNUM)

{

// Arrange

TPNumber a(**2.0**, **10**, **2**);

**double** expected = **2.0**;

// Act

**double** result = a.getNumber();

// Assert

Assert::AreEqual(expected, result);

}

TEST\_METHOD(GETNUMST)

{

// Arrange

TPNumber number(**15**, **16**, **2**);

std::string expected = "F";

// Act

std::string result = number.getNumberString();

// Assert

Assert::AreEqual(expected, result);

}

TEST\_METHOD(GETNUMST\_WithRightPartZeroBreak)

{

// Arrange

TPNumber number(**25.125**, **10**, **5**);

std::string expected = "25.125";

// Act

std::string result = number.getNumberString();

// Assert

Assert::AreEqual(expected, result);

}

TEST\_METHOD(GETNUMST\_WithIntegerNumber)

{

// Arrange

TPNumber number(**25.0**, **10**, **5**);

std::string expected = "25";

// Act

std::string result = number.getNumberString();

// Assert

Assert::AreEqual(expected, result);

}

TEST\_METHOD(GETBASE\_1)

{

// Arrange

TPNumber a(**10**, **16**, **2**);

**int** expected = **16**;

// Act

**int** result = a.getBase();

// Assert

Assert::AreEqual(expected, result);

}

TEST\_METHOD(GETBASESTR\_1)

{

// Arrange

TPNumber a(**10**, **16**, **2**);

std::string expected = "16";

// Act

std::string result = a.getBaseString();

// Assert

Assert::AreEqual(expected, result);

}

TEST\_METHOD(GETPR\_1)

{

// Arrange

TPNumber a(**10**, **16**, **2**);

**int** expected = **2**;

// Act

**int** result = a.getPrecision();

// Assert

Assert::AreEqual(expected, result);

}

TEST\_METHOD(GETPRSTR\_1)

{

// Arrange

TPNumber a(**10**, **16**, **2**);

std::string expected = "2";

// Act

std::string result = a.getPrecisionString();

// Assert

Assert::AreEqual(expected, result);

}

TEST\_METHOD(SET\_BASE\_1)

{

// Arrange

TPNumber a(**10**, **16**, **2**);

**int** expected = **5**;

// Act

a.setBase(expected);

// Assert

Assert::AreEqual(expected, a.getBase());

}

TEST\_METHOD(SET\_BASE\_STR\_1)

{

// Arrange

TPNumber a(**10**, **16**, **2**);

std::string baseStr = "5";

**int** expected = **5**;

// Act

a.setBase(baseStr);

// Assert

Assert::AreEqual(expected, a.getBase());

}

TEST\_METHOD(SET\_PR\_1)

{

// Arrange

TPNumber a(**10**, **16**, **2**);

**int** expected = **3**;

// Act

a.setPrecision(expected);

// Assert

Assert::AreEqual(expected, a.getPrecision());

}

TEST\_METHOD(SET\_PR\_STR\_1)

{

// Arrange

TPNumber a(**10**, **16**, **2**);

std::string precisionStr = "3";

**int** expected = **3**;

// Act

a.setPrecision(precisionStr);

// Assert

Assert::AreEqual(expected, a.getPrecision());

}

};

}