

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

**Лабораторная работа №5**  
**«Абстрактный тип данных (ADT) р - ичное число»**

Выполнил:

Студент группы ИП-113

Шпилев Д. И.

Работу проверил:

старший преподаватель кафедры ПМиК

Агалаков А.А.

Новосибирск 2024 г.

## Содержание

1. Задание.....	3
2. Исходный код программы .....	10
2.1. Код программы .....	10
2.2. Код тестов.....	17
3. Результаты модульных тестов.....	24
4. Вывод.....	26

## 1. Задание

### 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 C2, используя средства модульного тестирования Visual Studio.
3. Если необходимо, предусмотрите возбуждение исключительных ситуаций.

Спецификация типа данных «р-ичное число».

### ADT TPNumber

### 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 – ( $c \geq 0$ ). 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».

<b>Конструктор Число</b>	
Вход:	Вещественное число ( $a$ ), основание системы счисления ( $b$ ), точность представления числа ( $c$ )
Предусловия:	Основание системы счисления ( $b$ ) должно принадлежать интервалу [2..16], точность представления числа $c \geq 0$ .

Процесс:	<p>Инициализирует поля объекта this p-ичное число: система счисления (b), точность представления (c). В поле (n) числа заносится (a).</p> <p>Например:</p> <p><math>TPNumber(a,3,3)</math> = число a в системе счисления 3 с тремя разрядами после троичной точки.</p> <p><math>TPNumber(a,3,2)</math> = число a в системе счисления 3 с двумя разрядами после троичной точки.</p>
Постусловия:	Объект инициализирован начальными значениями.
Выход:	Нет.
<b>КонструкторСтрока</b>	
Вход:	Строковые представления: p-ичного числа (a), основания системы счисления (b), точности представления числа (c)
Предусловия:	Основание системы счисления (b) должно принадлежать интервалу [2..16], точность представления числа c $\geq 0$ .
Процесс:	<p>Инициализирует поля объекта this p-ичное число: основание системы счисления (b), точностью представления (c). В поле (n) числа this заносится результат преобразования строки (a) в числовое представление. b-ичное число (a) и основание системы счисления (b) представлены в формате строки.</p> <p>Например:</p> <p><math>TPNumber("20","3","6") = 20</math> в системе счисления 3, точность 6 знаков после запятой.</p>

	TPNumber ("0","3","8") = 0 в системе счисления 3, точность 8 знаков после запятой.
Постусловия:	Объект инициализирован начальными значениями.
Выход:	Нет.
<b>Копировать:</b>	
Вход:	Нет.
Предусловия:	Нет.
Процесс:	Создаёт копию самого числа this (тип TPNumber).
Выход:	р-ичное число.
Постусловия:	Нет.
<b>Сложить</b>	
Вход:	Р-ичное число d с основанием и точностью такими же, как у самого числа this.
Предусловия:	Нет.
Процесс:	Создаёт и возвращает р-ичное число (тип TPNumber), полученное сложением полей (n) самого числа this и числа d.
Выход:	р-ичное число.
Постусловия:	Нет
<b>Умножить</b>	
Вход:	Р-ичное число d с основанием и точностью такими же, как у самого числа this.
Предусловия:	Нет.
Процесс:	Создаёт и возвращает р-ичное число (тип TPNumber), полученное умножением полей (n)

	самого числа this и числа d.
Выход:	P-ичное число (тип TPNumber).
Постусловия:	Нет.
<b>Вычитать</b>	
Вход:	P-ичное число d с основанием и точностью такими же, как у самого числа this.
Предусловия:	Нет.
Процесс:	Создаёт и возвращает p-ичное число (тип TPNumber), полученное вычитанием полей (n) самого числа this и числа d.
Выход:	P-ичное число (тип TPNumber).
Постусловия:	Нет.
<b>Делить</b>	
Вход:	P-ичное число d с основанием и точностью такими же, как у самого числа.
Предусловия:	Поле (n) числа (d) не равно 0.
Процесс:	Создаёт и возвращает p-ичное число (тип TPNumber), полученное делением полей (n) самого числа this на поле (n) числа d.
Выход:	P-ичное число (тип TPNumber).
Постусловия:	Нет.
<b>Обратить</b>	
Вход:	Нет.
Предусловия:	Поле (n) самого числа не равно 0.
Процесс:	Создаёт p-ичное число, в поле (n) которого заносится значение, полученное как $1/(n)$ самого

	числа this.
Выход:	P-ичное число (тип TPNumber).
Постусловия:	Нет.
<b>Квадрат</b>	
Вход:	Нет.
Предусловия:	Нет.
Процесс:	Создаёт p-ичное число, в поле (n) которого заносится значение, полученное как квадрат поля (n) самого числа this.
Выход:	P-ичное число (тип TPNumber).
Постусловия:	Нет.
<b>ВзятьPЧисло</b>	
Вход:	Нет.
Предусловия:	Нет.
Процесс:	Возвращает значение поля (n) самого числа this.
Выход:	Вещественное значение.
Постусловия:	Нет.
<b>ВзятьPСтрока</b>	
Вход:	Нет.
Предусловия:	Нет.
Процесс:	Возвращает p-ичное число (q) в формате строки, изображающей значение поля (n) самого числа this в системе счисления (b) с точностью (c).
Выход:	Строка.
Постусловия:	Нет.

<b><i>ВзятьОснованиеЧисло</i></b>	
Вход:	Нет.
Предусловия:	Нет.
Процесс:	Возвращает значение поля (b) самого числа this
Выход:	Целочисленное значение
Постусловия:	Нет.
<b><i>ВзятьОснованиеСтрока</i></b>	
Вход:	Нет.
Предусловия:	Нет.
Процесс:	Возвращает значение поля (b) самого числа this в формате строки, изображающей (b) в десятичной системе счисления.
Выход:	Строка.
Постусловия:	Нет.
<b><i>ВзятьТочностьЧисло</i></b>	
Вход:	Нет.
Предусловия:	Нет.
Процесс:	Возвращает значение поля (c) самого числа this.
Выход:	Целое значение.
Постусловия:	Нет.
<b><i>ВзятьТочностьСтрока</i></b>	
Вход:	Нет.
Предусловия:	Нет.
Процесс:	Возвращает значение поля (c) самого числа this в формате строки, изображающей (c) в десятичной системе счисления.



Выход:	Строка.
Постусловия:	Нет.
<b>УстановитьОснованиеЧисло</b>	
Вход:	Целое число (newb).
Предусловия:	$2 \leq \text{newb} \leq 16$ .
Процесс:	Устанавливает в поле (b) самого числа this значение (newb).
Выход:	Нет.
Постусловия:	Нет.
<b>УстановитьОснованиеСтрока</b>	
Вход:	Строка (bs), изображающая основание (b) p-ичного числа в десятичной системе счисления.
Предусловия:	Допустимый диапазон числа, изображаемого строкой (bs) - 2,16.
Процесс:	Устанавливает значение поля (b) самого числа this значением, полученным в результате преобразования строки (bs).
Выход:	Строка.
Постусловия:	Нет.
<b>УстановитьТочностьЧисло</b>	
Вход:	Целое число (newc).
Предусловия:	$\text{newc} \geq 0$ .
Процесс:	Устанавливает в поле (c) самого числа значение (newc).
Выход:	Нет.
Постусловия:	Нет.
<b>УстановитьТочностьСтрока</b>	
Вход:	Строка (newc).
Предусловия:	Строка (newc) изображает десятичное целое $\geq 0$ .
Процесс:	Устанавливает в поле (c) самого числа this значение, полученное преобразованием строки (newc).
Выход:	Нет.
Постусловия:	Нет.

*end TPNumber*

## 2. Исходный код программы

### 2.1. Код программы

#### TPNumber.h

```
#pragma once
#include <string>
#include <stdexcept>
#include <cctype>

namespace STP {

    class TPNumber
    {
    public:
        TPNumber(double value, int base, int precision);
        TPNumber(const std::string &value, const std::string &base, const
std::string &precision);

        virtual ~TPNumber() = 0;

        virtual TPNumber* operator+(const TPNumber& d) const = 0;
        virtual TPNumber* operator-(const TPNumber& d) const = 0;
        virtual TPNumber* operator*(const TPNumber& d) const = 0;
        virtual TPNumber* operator/(const TPNumber& d) const = 0;

        virtual TPNumber* Invert() const = 0;
        virtual TPNumber* Square() const noexcept = 0;

        void setBase(const std::string& base);
        void setBase(const int& base);
        void setPrecision(const std::string& precision);
        void setPrecision(const int& precision);

        double number() const noexcept { return number_; }
        std::string numberString() const noexcept { return numberString_; }
        int base() const noexcept { return base_; }
        std::string baseString() const noexcept { return std::to_string(base_); }
    }

        int precision() const noexcept { return precision_; }
        std::string precisionString() const noexcept { return
std::to_string(precision_); }

    protected:
        std::string numberString_;
        double number_;
        int base_;
        int precision_;
    private:
        std::string ConvertToBase(const double &value, const int &base, int
precision) const noexcept;
        double StringToDouble(const std::string &value, const int &base) const;
    };
}
```

## TPNumber.cpp

```
#include "TPNumber.h"
#include <stdexcept>
#include <cmath>

namespace STP {

    TPNumber::TPNumber(double value, int base, int precision)
        : base_(base), precision_(precision) {

        if (base < 2 || base > 16) {
            throw std::invalid_argument("Base must be in the range
[2..16]");
        }
        if (precision < 0) {
            throw std::invalid_argument("Precision must be non-negative");
        }

        number_ = value;
        numberString_ = ConvertToBase(value, base, precision);
    }

    TPNumber::TPNumber(const std::string& value, const std::string& base, const
std::string& precision)
    {
        int baseInt, precisionInt;

        try {
            baseInt = std::stoi(base);
            precisionInt = std::stoi(precision);
        }
        catch (const std::invalid_argument&) {
            throw std::invalid_argument("Invalid base or precision string");
        }

        if (baseInt < 2 || baseInt > 16) {
            throw std::invalid_argument("Base must be in the range
[2..16]");
        }
        if (precisionInt < 0) {
            throw std::invalid_argument("Precision must be non-negative");
        }

        base_ = baseInt;
        precision_ = precisionInt;
        number_ = StringToDouble(value, baseInt);
        numberString_ = value;
    }

    TPNumber::~TPNumber() = default;

    std::string TPNumber::ConvertToBase(const double& value, const int& base, int
precision) const noexcept
    {
        std::string result;

        double absValue = std::fabs(value);
        long long intPart = static_cast<long long>(absValue);
        double fracPart = absValue - intPart;

        if (intPart == 0) {
            result = "0";
        }
        else {
```

```

        while (intPart > 0) {
            short digit = intPart % base;
            result.insert(0, 1, digit < 10 ? static_cast<char>('0' +
digit) : static_cast<char>('A' + digit - 10));
            intPart /= base;
        }
    }

    if (precision > 0) {
        result += ".";
        while (precision-- > 0) {
            fracPart *= base;
            short digit = static_cast<long long>(fracPart);
            fracPart -= digit;
            result += (digit < 10 ? static_cast<char>('0' + digit) :
static_cast<char>('A' + digit - 10));
        }
        if (value < 0) result.insert(0, 1, '-');
        return result;
    }

    double TPNNumber::StringToDouble(const std::string& value, const int& base)
const
    {
        double result = 0.0;
        bool isFraction = false;
        double fractionMultiplier = 1.0;
        bool isNegative = (value[0] == '-');

        size_t startIndex = (isNegative || value[0] == '+') ? 1 : 0;

        for (size_t i = startIndex; i < value.size(); ++i) {
            char ch = value[i];

            if (ch == '.') {
                isFraction = true;
                continue;
            }

            char upperCh = std::toupper(ch);

            short digit;
            if (upperCh >= '0' && upperCh <= '9') {
                digit = upperCh - '0';
            }
            else if (upperCh >= 'A' && upperCh <= 'F') {
                digit = upperCh - 'A' + 10;
            }
            else {
                throw std::invalid_argument("Invalid character in the
number string");
            }

            if (digit >= base) {
                throw std::invalid_argument("Digit out of range for the
specified base");
            }

            if (isFraction) {
                fractionMultiplier /= base;
                result += digit * fractionMultiplier;
            }
            else {
                result = result * base + digit;
            }
        }
    }

```

```

        }
    }

    return isNegative ? -result : result;
}

void TPNNumber::setBase(const std::string& base)
{
    int newBase;
    try {
        newBase = std::stoi(base);
        if (newBase < 2 || newBase > 16) {
            throw std::out_of_range("Base must be between 2 and 16");
        }
    }
    catch (const std::invalid_argument&) {
        throw std::invalid_argument("Invalid argument: the string is not
a valid integer");
    }
    catch (const std::out_of_range&) {
        throw std::out_of_range("The integer value is out of range");
    }

    base_ = newBase;
    numberString_ = ConvertToBase(number_, base_, precision_);
}

void TPNNumber::setBase(const int& base)
{
    if (base < 2 || base > 16) {
        throw std::out_of_range("Base must be between 2 and 16");
    }
    base_ = base;
    numberString_ = ConvertToBase(number_, base_, precision_);
}

void TPNNumber::setPrecision(const std::string& precision)
{
    int newPrecision;
    try {
        newPrecision = std::stoi(precision);
    }
    catch (const std::invalid_argument&) {
        throw std::invalid_argument("Invalid argument: the string is not
a valid integer");
    }
    if (newPrecision < 0) {
        throw std::invalid_argument("Precision must be non-negative");
    }
    precision_ = newPrecision;
    numberString_ = ConvertToBase(number_, base_, precision_);
}

void TPNNumber::setPrecision(const int& precision)
{
    if (precision < 0) {
        throw std::invalid_argument("Precision must be non-negative");
    }
    precision_ = precision;
    numberString_ = ConvertToBase(number_, base_, precision_);
}
}

```

## TPNumber.h

```
#pragma once
#include "TPNumber.h"

namespace STP {

    class PNumber :
        public TPNumber
    {
    public:
        PNumber(double value, int base, int precision)
            : TPNumber(value, base, precision) {}

        PNumber(const std::string& value, const std::string& base, const std::string
&precision)
            : TPNumber(value, base, precision) {}

        virtual TPNumber* operator+(const TPNumber& d) const override;
        virtual TPNumber* operator-(const TPNumber& d) const override;
        virtual TPNumber* operator*(const TPNumber& d) const override;
        virtual TPNumber* operator/(const TPNumber& d) const override;
        virtual PNumber operator+(const PNumber& d) const;
        virtual PNumber operator-(const PNumber& d) const;
        virtual PNumber operator*(const PNumber& d) const;
        virtual PNumber operator/(const PNumber& d) const;

        virtual TPNumber* Invert() const override;
        virtual TPNumber* Square() const noexcept override;

        ~PNumber() override = default;

    };
}
```

## TPNumber.cpp

```
#include "PNumber.h"

namespace STP {

    TPNumber* PNumber::operator+(const TPNumber& d) const
    {
        const PNumber* pNumberD = dynamic_cast<const PNumber*>(&d);
        if (pNumberD == nullptr) {
            throw std::invalid_argument("Operands must be of type PNumber");
        }

        if (base_ != pNumberD->base_ || precision_ != pNumberD->precision_) {
            throw std::invalid_argument("Bases and precisions must match");
        }

        double resultValue = number_ + pNumberD->number_;
        return new PNumber(resultValue, base_, precision_);
    }

    TPNumber* PNumber::operator-(const TPNumber& d) const
    {
        const PNumber* pNumberD = dynamic_cast<const PNumber*>(&d);
        if (pNumberD == nullptr) {
            throw std::invalid_argument("Operands must be of type PNumber");
        }
    }
}
```

```

    if (base_ != pNumberD->base_ || precision_ != pNumberD->precision_) {
        throw std::invalid_argument("Bases and precisions must match");
    }

    double resultValue = number_ - pNumberD->number_;
    return new PNumber(resultValue, base_, precision_);
}

TPNumber* PNumber::operator*(const TPNumber& d) const
{
    const PNumber* pNumberD = dynamic_cast<const PNumber*>(&d);
    if (pNumberD == nullptr) {
        throw std::invalid_argument("Operands must be of type PNumber");
    }

    if (base_ != pNumberD->base_ || precision_ != pNumberD->precision_) {
        throw std::invalid_argument("Bases and precisions must match");
    }

    double resultValue = number_ * pNumberD->number_;
    return new PNumber(resultValue, base_, precision_);
}

TPNumber* PNumber::operator/(const TPNumber& d) const
{
    const PNumber* pNumberD = dynamic_cast<const PNumber*>(&d);
    if (pNumberD == nullptr) {
        throw std::invalid_argument("Operands must be of type PNumber");
    }

    if (base_ != pNumberD->base_ || precision_ != pNumberD->precision_) {
        throw std::invalid_argument("Bases and precisions must match");
    }

    if (pNumberD->number_ == 0.0) {
        throw std::invalid_argument("Division by zero");
    }

    double resultValue = number_ / pNumberD->number_;
    return new PNumber(resultValue, base_, precision_);
}

PNumber PNumber::operator+(const PNumber& d) const
{
    if (base_ != d.base_ || precision_ != d.precision_) {
        throw std::invalid_argument("Bases and precisions must match");
    }
    double resultValue = number_ + d.number_;
    return PNumber(resultValue, base_, precision_);
}

PNumber PNumber::operator-(const PNumber& d) const
{
    if (base_ != d.base_ || precision_ != d.precision_) {
        throw std::invalid_argument("Bases and precisions must match");
    }
    double resultValue = number_ - d.number_;
    return PNumber(resultValue, base_, precision_);
}

PNumber PNumber::operator*(const PNumber& d) const
{
    if (base_ != d.base_ || precision_ != d.precision_) {
        throw std::invalid_argument("Bases and precisions must match");
    }

```

```

    }
    double resultValue = number_ * d.number_;
    return PNumber(resultValue, base_, precision_);
}

PNumber PNumber::operator/(const PNumber& d) const
{
    if (base_ != d.base_ || precision_ != d.precision_) {
        throw std::invalid_argument("Bases and precisions must match");
    }

    if (d.number_ == 0.0) {
        throw std::invalid_argument("Division by zero");
    }
    double resultValue = number_ / d.number_;
    return PNumber(resultValue, base_, precision_);
}

TPNumber* PNumber::Invert() const
{
    if (number_ == 0.0) {
        throw std::invalid_argument("Cannot invert zero");
    }

    double resultValue = 1.0 / number_;
    return new PNumber(resultValue, base_, precision_);
}

TPNumber* PNumber::Square() const noexcept
{
    double resultValue = number_ * number_;
    return new PNumber(resultValue, base_, precision_);
}
}

```



## 2.2.Код тестов

### TPNumberTest.cs

```
#include "CppUnitTest.h"
#include "../Lab5/PNumber.h"

using namespace STP;
using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace TPNumberTest
{
    TEST_CLASS(TPNumberTest)
    {
    public:

        TEST_METHOD(ConstructorDoubleBasePrecision)
        {
            PNumber num1(10.0, 10, 2);
            Assert::AreEqual(10.0, num1.number());
            Assert::AreEqual(10, num1.base());
            Assert::AreEqual(2, num1.precision());
            PNumber num2(234.153, 5, 8);
            Assert::AreEqual(std::string("1414.03403030"),
num2.numberString());
            Assert::AreEqual(5, num2.base());
            Assert::AreEqual(8, num2.precision());
        }

        TEST_METHOD(ConstructorDoubleBasePrecisionExceptions)
        {
            Assert::ExpectException<std::invalid_argument>([]
{PNumber num1(10.0, 20, 2); });
            Assert::ExpectException<std::invalid_argument>([]
{PNumber num1(10.0, 1, 2); });
            Assert::ExpectException<std::invalid_argument>([]
{PNumber num1(10.0, 6, -1); });
        }

        TEST_METHOD(ConstructorStringBasePrecisionExceptions)
        {
            Assert::ExpectException<std::invalid_argument>([]
{PNumber num1("10.0", "20", "2"); });
            Assert::ExpectException<std::invalid_argument>([]
{PNumber num1("10.0", "1", "2"); });
            Assert::ExpectException<std::invalid_argument>([]
{PNumber num1("10.0", "6", "-1"); });
            Assert::ExpectException<std::invalid_argument>([]
{PNumber num1("1~0.0", "6", "1"); });
        }
    }
}
```

```

        Assert::ExpectException<std::invalid_argument>([
{PNumber num1("80.0", "6", "1"); });
        Assert::ExpectException<std::invalid_argument>([
{PNumber num1("10.0", "Z", "2"); });

    }

    TEST_METHOD(ConstructorStringBasePrecision)
    {
        PNumber num2("-A.B9", "16", "4");
        Assert::AreEqual(-10.7227, num2.number(), 0.0001);
        Assert::AreEqual(16, num2.base());
        Assert::AreEqual(4, num2.precision());
    }

    TEST_METHOD(SetBaseFromInt)
    {
        PNumber num1(10.0, 10, 2);
        num1.setBase(2);
        Assert::AreEqual(2, num1.base());
    }

    TEST_METHOD(SetBaseFromIntExceptions)
    {
        PNumber num1("10.0", "3", "2");
        Assert::ExpectException<std::out_of_range>([&] {
num1.setBase(20); });
        Assert::ExpectException<std::out_of_range>([&] {
num1.setBase(1); });
    }

    TEST_METHOD(SetBaseFromString)
    {
        PNumber num1(10.0, 10, 2);
        num1.setBase("8");
        Assert::AreEqual(8, num1.base());
    }

    TEST_METHOD(SetBaseFromStringExceptions)
    {
        PNumber num1("10.0", "3", "2");
        Assert::ExpectException<std::out_of_range>([&] {
num1.setBase("20"); });
        Assert::ExpectException<std::out_of_range>([&] {
num1.setBase("1"); });
        Assert::ExpectException<std::invalid_argument>([&] {
num1.setBase("q`sdfa"); });
    }

```

```

TEST_METHOD(SetPrecisionFromInt)
{
    PNumber num1(10.0, 10, 2);
    num1.setPrecision(4);
    Assert::AreEqual(4, num1.precision());
}

TEST_METHOD(SetPrecisionFromIntExceptions)
{
    PNumber num1("10.0", "3", "2");
    Assert::ExpectException<std::invalid_argument>([&] {
num1.setPrecision(-1); });
}

TEST_METHOD(SetPrecisionFromString)
{
    PNumber num1(10.0, 10, 2);
    num1.setPrecision("5");
    Assert::AreEqual(5, num1.precision());
}

TEST_METHOD(SetPrecisionFromStringExceptions)
{
    PNumber num1("10.0", "3", "2");
    Assert::ExpectException<std::invalid_argument>([&] {
num1.setPrecision("-1"); });
    Assert::ExpectException<std::invalid_argument>([&] {
num1.setPrecision("`dsgfa"); });
}

TEST_METHOD(Addition)
{
    PNumber num1(10.0, 10, 2);
    PNumber num4(20.0, 10, 2);
    PNumber result = num1 + num4;
    Assert::AreEqual(30.0, result.number());
    Assert::AreEqual(10, result.base());
    Assert::AreEqual(2, result.precision());
}

TEST_METHOD(OperationsExceptinsBase)
{
    PNumber num1(10.0, 10, 2);
    PNumber num4(20.0, 12, 2);
    Assert::ExpectException<std::invalid_argument>([&] {
PNumber result = num1 + num4; });
    Assert::ExpectException<std::invalid_argument>([&] {
PNumber result = num1 - num4; });
}

```

```

        Assert::ExpectException<std::invalid_argument>([&] {
PNumber result = num1 * num4; });
        Assert::ExpectException<std::invalid_argument>([&] {
PNumber result = num1 / num4; });
    }

    TEST_METHOD(OperationsExceptinsPrecision)
    {
        PNumber num1(10.0, 10, 4);
        PNumber num4(20.0, 10, 2);
        Assert::ExpectException<std::invalid_argument>([&] {
PNumber result = num1 + num4; });
        Assert::ExpectException<std::invalid_argument>([&] {
PNumber result = num1 - num4; });
        Assert::ExpectException<std::invalid_argument>([&] {
PNumber result = num1 * num4; });
        Assert::ExpectException<std::invalid_argument>([&] {
PNumber result = num1 / num4; });
    }

    TEST_METHOD(OperationsExceptinDivisionByZero)
    {
        PNumber num1(10.0, 10, 4);
        PNumber num4(0.0, 10, 4);
        Assert::ExpectException<std::invalid_argument>([&] {
PNumber result = num1 / num4; });
    }

    TEST_METHOD(Subtraction)
    {
        PNumber num1(10.0, 10, 2);
        PNumber num4(20.0, 10, 2);
        PNumber result = num4 - num1;
        Assert::AreEqual(10.0, result.number());
        Assert::AreEqual(10, result.base());
        Assert::AreEqual(2, result.precision());
    }

    TEST_METHOD(Multiplication)
    {
        PNumber num1(10.0, 10, 2);
        PNumber num4(20.0, 10, 2);
        PNumber result = num1 * num4;
        Assert::AreEqual(200.0, result.number());
        Assert::AreEqual(10, result.base());
        Assert::AreEqual(2, result.precision());
    }

    TEST_METHOD(Division)

```

```

    {
        PNumber num1(10.0, 10, 2);
        PNumber num4(20.0, 10, 2);
        PNumber result = num4 / num1;
        Assert::AreEqual(2.0, result.number());
        Assert::AreEqual(10, result.base());
        Assert::AreEqual(2, result.precision());
    }

TEST_METHOD(AdditionRef)
{
    TPNumber* num1 = new PNumber(10.0, 10, 2);
    TPNumber *num4 = new PNumber(20.0, 10, 2);
    TPNumber *result = *num1 + *num4;
    Assert::AreEqual(30.0, result->number());
    Assert::AreEqual(10, result->base());
    Assert::AreEqual(2, result->precision());
}

TEST_METHOD(OperationsExceptinsBaseRef)
{
    TPNumber* num1 = new PNumber(10.0, 4, 2);
    TPNumber* num4 = new PNumber(20.0, 10, 2);
    Assert::ExpectException<std::invalid_argument>([&] {
TPNumber *result = *num1 + *num4; });
    Assert::ExpectException<std::invalid_argument>([&] {
TPNumber *result = *num1 - *num4; });
    Assert::ExpectException<std::invalid_argument>([&] {
TPNumber *result = *num1 * *num4; });
    Assert::ExpectException<std::invalid_argument>([&] {
TPNumber *result = *num1 / *num4; });
}

TEST_METHOD(OperationsExceptinsPrecisionRef)
{
    TPNumber* num1 = new PNumber(10.0, 10, 4);
    TPNumber* num4 = new PNumber(20.0, 10, 2);
    Assert::ExpectException<std::invalid_argument>([&] {
TPNumber* result = *num1 + *num4; });
    Assert::ExpectException<std::invalid_argument>([&] {
TPNumber* result = *num1 - *num4; });
    Assert::ExpectException<std::invalid_argument>([&] {
TPNumber* result = *num1 * *num4; });
    Assert::ExpectException<std::invalid_argument>([&] {
TPNumber* result = *num1 / *num4; });
}

TEST_METHOD(OperationsExceptinDivisionByZeroRef)
{

```

```

        TPNumber* num1 = new PNumber(10.0, 10, 4);
        TPNumber* num4 = new PNumber(0.0, 10, 4);
        Assert::ExpectException<std::invalid_argument>([&] {
TPNumber* result = *num1 / *num4; });
    }

TEST_METHOD(SubtractionRef)
{
    TPNumber* num1 = new PNumber(10.0, 10, 2);
    TPNumber* num4 = new PNumber(20.0, 10, 2);
    TPNumber* result = *num1 - *num4;
    Assert::AreEqual(-10.0, result->number());
    Assert::AreEqual(10, result->base());
    Assert::AreEqual(2, result->precision());
}

TEST_METHOD(MultiplicationRef)
{
    TPNumber* num1 = new PNumber(10.0, 10, 2);
    TPNumber* num4 = new PNumber(20.0, 10, 2);
    TPNumber* result = *num1 * *num4;
    Assert::AreEqual(200.0, result->number());
    Assert::AreEqual(10, result->base());
    Assert::AreEqual(2, result->precision());
}

TEST_METHOD(DivisionRef)
{
    TPNumber* num1 = new PNumber(10.0, 10, 2);
    TPNumber* num4 = new PNumber(20.0, 10, 2);
    TPNumber* result = *num4 / *num1;
    Assert::AreEqual(2.0, result->number());
    Assert::AreEqual(10, result->base());
    Assert::AreEqual(2, result->precision());
}

TEST_METHOD(Invert)
{
    TPNumber *num1 = new PNumber(10.0, 10, 2);
    TPNumber* result = num1->Invert();
    Assert::AreEqual(0.1, result->number());
    Assert::AreEqual(10, result->base());
    Assert::AreEqual(2, result->precision());
    delete num1;
    delete result;
}

TEST_METHOD(InvertException)
{

```

```

        TPNumber* num1 = new PNumber(0.0, 10, 2);
        Assert::ExpectException<std::invalid_argument>([&] {
TPNumber* result = num1->Invert(); });
    }

    TEST_METHOD(Square)
    {
        TPNumber* num1 = new PNumber(10.0, 10, 2);
        TPNumber* result = num1->Square();
        Assert::AreEqual(100.0, result->number());
        Assert::AreEqual(10, result->base());
        Assert::AreEqual(2, result->precision());
        delete num1;
        delete result;
    }

    TEST_METHOD(DivisionByZero)
    {
        TPNumber* num1 = new PNumber(10.0, 10, 2);
        TPNumber* num4 = new PNumber(0.0, 10, 2);
        Assert::ExpectException<std::invalid_argument>([&]
{TPNumber* result = *num1 / *num4; });
        delete num1;
        delete num4;
    }
};
}

```

### 3. Результаты модульных тестов

Обозреватель тестов			
<div> <span>▶</span> <span>▶</span> <span>↺</span> <span>✖</span> <span>⚙</span> <span>30</span> <span>30</span> <span>0</span> <span>📄</span> <span>[≡]</span> <span>+</span> <span>-</span> <span>⚙</span> </div>			
Запуск тестов завершен: тестов запущено в 121 мс: 30 (пройдено: 30, не пройдено: 0, пропущено: 0).			
Тестирование	Длительн...	Признаки	Сообщение об ошибке
▲ ✓ TPNumberTest (30)	< 1 мс		
▲ ✓ TPNumberTest (30)	< 1 мс		
▲ ✓ TPNumberTest (30)	< 1 мс		
✓ Addition	< 1 мс		
✓ AdditionRef	< 1 мс		
✓ ConstructorDoubleBasePrecision	< 1 мс		
✓ ConstructorDoubleBasePrecision...	< 1 мс		
✓ ConstructorStringBasePrecision	< 1 мс		
✓ ConstructorStringBasePrecision...	< 1 мс		
✓ Division	< 1 мс		
✓ DivisionByZero	< 1 мс		
✓ DivisionRef	< 1 мс		
✓ Invert	< 1 мс		
✓ InvertException	< 1 мс		
✓ Multiplication	< 1 мс		
✓ MultiplicationRef	< 1 мс		
✓ OperationsExceptinDivisionByZ...	< 1 мс		
✓ OperationsExceptinDivisionByZ...	< 1 мс		
✓ OperationsExceptinsBase	< 1 мс		
✓ OperationsExceptinsBaseRef	< 1 мс		
✓ OperationsExceptinsPrecision	< 1 мс		
✓ OperationsExceptinsPrecisionRef	< 1 мс		
✓ SetBaseFromInt	< 1 мс		
✓ SetBaseFromIntExceptions	< 1 мс		
✓ SetBaseFromString	< 1 мс		
✓ SetBaseFromStringExceptions	< 1 мс		
✓ SetPrecisionFromInt	< 1 мс		
✓ SetPrecisionFromIntExceptions	< 1 мс		
✓ SetPrecisionFromString	< 1 мс		
✓ SetPrecisionFromStringExceptio...	< 1 мс		
✓ Square	< 1 мс		
✓ Subtraction	< 1 мс		
✓ SubtractionRef	< 1 мс		



Hierarchy	Covered (%Blocks)	Not Covered (%Blocks)	Covered (%Lines)	Not Covered (%Lines)
d_shp_KASPENIUM_2024-09-13.16_26_03.coverage	90,99%	9,01%	83,46%	2,01%
tpnumbertest.dll	90,99%	9,01%	83,46%	2,01%
{ } STP	85,11%	14,89%	78,97%	4,10%
PNumber	78,99%	21,01%	73,42%	5,06%
PNumber	100,00%	0,00%	100,00%	0,00%
~PNumber	100,00%	0,00%	100,00%	0,00%
operator*	87,50%	12,50%	83,33%	0,00%
operator*	70,59%	29,41%	66,67%	11,11%
operator-	87,50%	12,50%	83,33%	0,00%
operator-	70,59%	29,41%	66,67%	11,11%
operator+	87,50%	12,50%	83,33%	0,00%
operator+	70,59%	29,41%	66,67%	11,11%
operator/	84,62%	15,38%	75,00%	0,00%
operator/	72,73%	27,27%	63,64%	9,09%
Invert	83,33%	16,67%	66,67%	0,00%
Square	85,71%	14,29%	75,00%	0,00%
PNumber	100,00%	0,00%	100,00%	0,00%
TPNumber	90,06%	9,94%	82,76%	3,45%
TPNumber	86,36%	13,64%	73,33%	6,67%
TPNumber	86,67%	13,33%	75,00%	0,00%
~TPNumber	100,00%	0,00%	100,00%	0,00%
ConvertToBase	93,75%	6,25%	91,67%	0,00%
StringToDouble	94,87%	5,13%	93,10%	0,00%
setBase	90,00%	10,00%	83,33%	0,00%
setBase	77,78%	22,22%	61,54%	15,38%
setPrecision	88,89%	11,11%	83,33%	0,00%
setPrecision	85,71%	14,29%	70,00%	10,00%
base	100,00%	0,00%	100,00%	0,00%
number	100,00%	0,00%	100,00%	0,00%
numberString	100,00%	0,00%	100,00%	0,00%
precision	100,00%	0,00%	100,00%	0,00%
{ } TPNumberTest	94,78%	5,22%	87,75%	0,00%

#### **4. Вывод**

По итогам данной лабораторной работе были сформированы практические навыки разработки на C++ и модульного тестирования классов средствами Visual Studio.