**ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ**

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

**«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ ИМ. ПРОФ. М.А. БОНЧ-БРУЕВИЧА»**

(СПбГУТ)

ФАКУЛЬТЕТ ИНФОКОММУНИКАЦИОННЫХ СЕТЕЙ И СИСТЕМ **(ИКСС)**

КАФЕДРА ПРОГРАММНОЙ ИНЖЕНЕРИИ И ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ **(ПИ И ВТ)**

Дисциплина: «ООП»

Лабораторная работа №5.

**Тема: «Шаблоны классов. Работа с исключительными»**

Вариант №16

Выполнил: Студент группы ИКПИ-95

Новиков С.С.

Подпись \_\_\_\_\_\_\_\_\_\_\_\_\_\_

Приняла:

Петрова О.Б.

Подпись \_\_\_\_\_\_\_\_\_\_\_\_\_\_

2020г

*Постановка задачи*

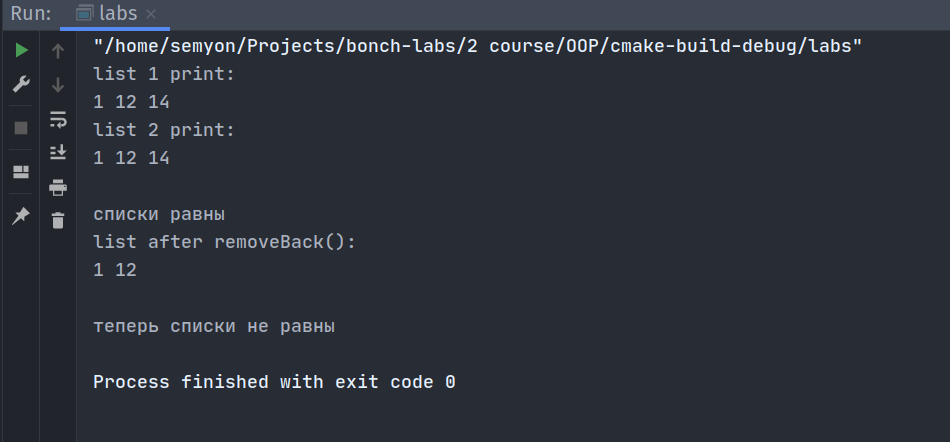
*Задача 1*

В настоящей лабораторной работе необходимо решить две задачи, связанные с организацией шаблонов классов. Первая из задач состоит в преобразовании в шаблон класс того числового класса, который был разработан студентом в первой лабораторной работе по ООП. Вторая задача состоит в разработке шаблона контейнера. При решении второй задачи следует предусмотреть обработку исключительных ситуаций.

*Задача 2*

Разработать шаблон класса для двунаправленного списка.

*Результаты работы программы*



*Программа на языке C****++***

Файл main.cpp

#include "list.hpp"

using namespace std;

int main() {

List<int> list;

list.pushBack(1);

List<int> list2(list);

list.pushBack(12);

list.pushBack(14);

list2.pushBack(12);

list2.pushBack(14);

cout << "list 1 print: " << endl;

list.print();

cout << endl;

cout << "list 2 print: " << endl;

list2.print();

cout << endl;

if (list == list2) {

cout << endl << "списки равны" << endl;

}

list.removeBack();

cout << "list after removeBack(): " << endl;

list.print();

cout << endl;

if (list != list2) {

cout << endl << "теперь списки не равны" << endl;

}

return 0;

}

Файл ComplexNumber.hpp

#pragma once

#include <iostream>

template<typename RealNum, typename ImaginaryNum>

class ComplexNum {

public:

ComplexNum();

ComplexNum(RealNum real, ImaginaryNum imaginary);

ComplexNum(const ComplexNum<RealNum, ImaginaryNum> &r);

ComplexNum(const ComplexNum<RealNum, ImaginaryNum> &&r) noexcept;

~ComplexNum();

RealNum getReal() const;

ImaginaryNum getImaginary() const;

// сложение комплексных чисел

ComplexNum operator+(const ComplexNum &r) const;

// вычитание комплексных чисел

ComplexNum operator-(const ComplexNum &r) const;

// умножение комплексных чисел

ComplexNum operator\*(const ComplexNum &r) const;

// деление комплексных чисел

ComplexNum operator/(const ComplexNum &r) const;

// оператор присваивания

ComplexNum &operator=(const ComplexNum &r);

// операторы сравнения

bool operator==(const ComplexNum<RealNum, ImaginaryNum> &r) const;

bool operator!=(const ComplexNum<RealNum, ImaginaryNum> &r) const;

RealNum real;

ImaginaryNum imaginary;

};

template<typename RealNum, typename ImaginaryNum>

std::ostream &operator<<(std::ostream &out, const ComplexNum<RealNum, ImaginaryNum> &r);

typedef ComplexNum<float, float> ComplexNumF;

typedef ComplexNum<double, double> ComplexNumD;

#include "ComplexNumber.inl"

Файл ComplexNumber.inl

#include "ComplexNumber.hpp"

// конструктор

template<typename RealNum, typename ImaginaryNum>

ComplexNum<RealNum, ImaginaryNum>::ComplexNum():

real(), imaginary() {

}

// конструктор с параметрами

template<typename RealNum, typename ImaginaryNum>

ComplexNum<RealNum, ImaginaryNum>::ComplexNum(RealNum real, ImaginaryNum imaginary) :

real(real), imaginary(imaginary) {

}

// конструктор копирования

template<typename RealNum, typename ImaginaryNum>

ComplexNum<RealNum, ImaginaryNum>::ComplexNum(const ComplexNum<RealNum, ImaginaryNum> &r) :

real(r.real), imaginary(r.imaginary) {

}

// конструктор перемещения

template<typename RealNum, typename ImaginaryNum>

ComplexNum<RealNum, ImaginaryNum>::ComplexNum(const ComplexNum<RealNum, ImaginaryNum> &&r) noexcept :

real(r.real), imaginary(r.imaginary) {

}

// деструктор

template<typename RealNum, typename ImaginaryNum>

ComplexNum<RealNum, ImaginaryNum>::~ComplexNum() = default;

// получить вещественную часть комплексного числа

template<typename RealNum, typename ImaginaryNum>

RealNum ComplexNum<RealNum, ImaginaryNum>::getReal() const {

return real;

}

// получить мнимую часть комплексного числа

template<typename RealNum, typename ImaginaryNum>

ImaginaryNum ComplexNum<RealNum, ImaginaryNum>::getImaginary() const {

return imaginary;

}

// оператор сложения

template<typename RealNum, typename ImaginaryNum>

ComplexNum<RealNum, ImaginaryNum>

ComplexNum<RealNum, ImaginaryNum>::operator+(const ComplexNum<RealNum, ImaginaryNum> &r) const {

auto c = \*this;

c.real += r.real;

c.imaginary += r.imaginary;

return c;

}

// оператор вычитания

template<typename RealNum, typename ImaginaryNum>

ComplexNum<RealNum, ImaginaryNum>

ComplexNum<RealNum, ImaginaryNum>::operator-(const ComplexNum<RealNum, ImaginaryNum> &r) const {

auto c = \*this;

c.real -= r.real;

c.imaginary -= r.imaginary;

return c;

}

// оператор умножения

template<typename RealNum, typename ImaginaryNum>

ComplexNum<RealNum, ImaginaryNum>

ComplexNum<RealNum, ImaginaryNum>::operator\*(const ComplexNum<RealNum, ImaginaryNum> &r) const {

auto c = \*this;

const RealNum temp = real \* r.real - static\_cast<RealNum>(imaginary \* r.imaginary);

c.imaginary = static\_cast<ImaginaryNum>(real) \* r.imaginary + imaginary \* static\_cast<ImaginaryNum>(r.real);

c.real = temp;

return c;

}

// оператор деления

template<typename RealNum, typename ImaginaryNum>

ComplexNum<RealNum, ImaginaryNum>

ComplexNum<RealNum, ImaginaryNum>::operator/(const ComplexNum<RealNum, ImaginaryNum> &r) const {

RealNum real\_t = real \* r.real + static\_cast<RealNum>(imaginary \* r.imaginary);

ImaginaryNum imag\_t = imaginary \* static\_cast<ImaginaryNum>(r.real) - static\_cast<ImaginaryNum>(real) \* r.imaginary;

real\_t /= r.real \* r.real + static\_cast<RealNum>(r.imaginary \* r.imaginary);

imag\_t /= static\_cast<ImaginaryNum>(r.real \* r.real) + r.imaginary \* r.imaginary;

auto c = \*this;

c.real = real\_t;

c.imaginary = imag\_t;

return c;

}

// оператор присваивания

template<typename RealNum, typename ImaginaryNum>

ComplexNum<RealNum, ImaginaryNum> &

ComplexNum<RealNum, ImaginaryNum>::operator=(const ComplexNum<RealNum, ImaginaryNum> &r) {

real = r.real;

imaginary = r.imaginary;

return \*this;

}

// операторы сравнения

template<typename RealNum, typename ImaginaryNum>

bool ComplexNum<RealNum, ImaginaryNum>::operator==(const ComplexNum<RealNum, ImaginaryNum> &r) const {

return (real == r.real) && (imaginary == r.imaginary);

}

template<typename RealNum, typename ImaginaryNum>

bool ComplexNum<RealNum, ImaginaryNum>::operator!=(const ComplexNum<RealNum, ImaginaryNum> &r) const {

return (real != r.real) || (imaginary != r.imaginary);

}

template<typename RealNum, typename ImaginaryNum>

std::ostream &operator<<(std::ostream &out, const ComplexNum<RealNum, ImaginaryNum> &r) {

out << "(" << r.real << ", " << r.imaginary << ")";

return out;

}

Файл list.hpp

#pragma once

#include <iostream>

using namespace std;

template<typename T>

class List {

public:

List();

List(const T \*arr, size\_t count);

List(const List &l);

~List();

void pushFront(const T &elem);

void pushBack(const T &elem);

void removeFront();

void removeBack();

void print();

bool operator==(const List &list);

bool operator!=(const List &list);

struct Elem {

Elem \*next;

Elem \*prev;

T data;

};

private:

Elem \*first;

Elem \*last;

};

template<typename T>

List<T>::List(): first(nullptr), last(nullptr) {

this->first = nullptr;

this->last = nullptr;

}

template<typename T>

List<T>::List(const T \*arr, size\_t count): first(nullptr), last(nullptr) {

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

pushBack(arr[i]);

}

}

template<typename T>

List<T>::List(const List &l) {

if (l.first == nullptr) {

return;

}

auto current = l.first;

while (current != nullptr) {

pushBack(current->data);

current = current->next;

}

}

template<typename T>

List<T>::~List() {

auto current = this->first;

while (current != nullptr) {

delete current->prev;

current = current->next;

}

}

template<typename T>

void List<T>::pushFront(const T &elem) {

if (this->first == nullptr) {

first = last = new Elem;

} else {

auto oldFirst = this->first;

first = new Elem;

first->next = oldFirst;

first->prev = nullptr;

oldFirst->prev = first;

}

first->data = elem;

}

template<typename T>

void List<T>::pushBack(const T &elem) {

if (this->last == nullptr) {

last = first = new Elem;

} else {

auto oldLast = this->last;

last = new Elem;

last->next = nullptr;

last->prev = oldLast;

oldLast->next = last;

}

last->data = elem;

}

template<typename T>

void List<T>::print() {

if (this->first == nullptr) {

return;

// throw "пустой список";

}

cout << this->first->data;

cout << " ";

auto current = this->first->next;

while (current != nullptr) {

cout << current->data << " ";

current = current->next;

}

}

template<typename T>

void List<T>::removeFront() {

if (this->first != nullptr) {

this->first = first->next;

delete this->first->prev;

this->first->prev = nullptr;

}

}

template<typename T>

void List<T>::removeBack() {

if (this->first != this->last) {

this->last = last->prev;

delete this->last->next;

this->last->next = nullptr;

} else {

delete this->first;

this->first == nullptr;

this->last == nullptr;

}

}

template<typename T>

bool List<T>::operator==(const List &list) {

Elem \*t1, \*t2;

t1 = this->first;

t2 = list.first;

while (t1 != nullptr && t2 != nullptr) {

if (t1->data != t2->data)

return false;

t1 = t1->next;

t2 = t2->next;

}

if (t2 != nullptr || t1 != nullptr) {

return false;

}

return true;

}

template<typename T>

bool List<T>::operator!=(const List &list) {

return !(\*this == list);

}

*Выводы*

В результате проведенной работы изучены классы в c++.