НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ЯДЕРНЫЙ УНИВЕРСИТЕТ «МИФИ»

ИИКС

Кафедра №22 «Кибернетика»

Лабораторная работа №2

Выполнил:

студент гр. Б17-514

Жарков М. А.

Преподаватель:

Трифоненков А.В.

Москва 2018

Вариант №9

Коллекция: Линейная форма

Типы хранимых данных: вещественные числа, комплексные числа

Операции над коллекцией: сложение , умножение на скаляр, вычисление значения при заданных значениях аргументов

Линейная форма – это многочлен первой степени от n переменных:
$F_n(x_1, \ldots, x_n) = a_0 + a_1 x_1 + \ldots + a_n x_n$. a_i - коэффициенты, x_i - аргументы

Сложение линейных форм – функция, аргументами которой являются две линейные формы, а результатом которой является новая линейная форма, i-ый коэффициент которой равен сумме i-ых коэффициентов первой и второй линейных форм.

Умножение на скаляр – функция, аргументами которой являются линейная форма и целое или вещественное число, а результатом которой является линейная форма коэффициенты которой равны произведению i-го коэффициента изначальной линейной форме и числа.

Вычисление значений при заданных аргументах – функция, аргументами которой является линейная форма и массив аргументов, а результатом которой является целое или вещественное число, равное сумме произведений i-го аргумента и i-го коэффициента линейной формы.

Техническое задание

Создать заголовочный файл с обьявлением класса списка **list.hpp** и класса комплексных **complex.cpp**

Объявить шаблон класса List со следующими полями и разработать реализацию методов этого шаблона:

Название поля	Описание
Private	
`Head`	`Указатель на первый элемент списка коэффициентов линейной формы`
`Length`	`Длина списка (т.е. размерность линейной формы)`
Public	
`List()`	`Конструктор`
`~List`	`Деструктор`
`List(const List &lst)`	`Конструктор копирования`
`int get_length();`	`Возвращает значение length`
`void push(T value);`	`Добавляет элемент в конец линейной формы`

Название поля	Описание
`void pop();`	Удаляет элемент из конца линейной формы
`void print();`	Выводит линейную форму в консоль
`void sum_with(List<T>* sec_list);`	Складывает две линейные формы
`void multiply_by(T cnst);`	Умножает элементы линейной формы на число
`void correct_zeros();`	Корректирует линейную форму (удаляет все нулевые элементы, находящиеся в конце списка)
`T operator[](int index);`	Находит значение коэффициента по индексу
`T calculate_value(List<T> x);`	Считает значение линейной формы по заданным аргументам

Так же объявляется шаблон структуры ListItem:

Название поля	Описание
Private	
`T item`	Значение коэффициента
`ListItem* next`	Указатель на следующий элемент списка
Public	
`T get_item()`	Возвращает значение коэффициента

И класс комплексных:

Название поля	Описание
Private	
`Re`	Действительная часть комплексного
`Im`	Мнимая часть комплексного
Public	
`Complex()`	Конструктор по умолчанию
`Complex (double)`	Конструктор
`Complex (double, double)`	Конструктор
`~Complex();`	Деструктор
`double return_re()`	Возвращает значение re

Название поля	Описание
`double return_im()`	Возвращает значение im
`Complex operator+ (Complex);` `Complex operator+= (Complex);`	Перегрузка операторов сложения с комплексным
`Complex operator* (Complex);`	Перегрузка оператора умножения на комплексное
`Complex operator+ (double);` `Complex operator+= (double);`	Перегрузка операторов сложения с вещественным
`Complex operator* (double);` `Complex operator*= (double);`	Перегрузка оператора умножения на вещественное
`bool operator== (Complex);` `bool operator!= (Complex);`	Перегрузка операторов сравнения с комплексным
`bool operator== (double);` `bool operator!= (double);`	Перегрузка операторов сравнения с вещественным

`friend std::ostream &`
`operator<<(std::ostream &os, const`
`Complex &c);`
	Перегрузка оператора вывода

Пользовательский интерфейс

В программе реализован консольный интерфейс, с помощью которого пользователь создает линейную форму и работает с ней. Для работы с пользовательским интерфейсом необходимо подключить файл **UI.hpp**, а для его запуска необходимо вызвать метод **run_UI().**

С помощью пользовательского интерфейс программа может выполнять следующие команды:

Номер команды	Результат выполнение команды
1	Создание линейной формы
2	Сложение двух линейных формы
3	Умножение линейной формы на число
4	Вычисление значения при заданных аргументах
7	Выход из программы

При создании линейной формы программа предлагает выбрать тип коэффициентов линейной формы (вещественные или комплексные), размерность линейной формы, а затем сами коэффициенты, после чего создает линейную форму основанную на списке по заданным данным.

При сложении линейных форм программа предлагает создать вторую линейную форму, после чего складывает их. Если не задана линейная форма программа выведет предупреждение в консоль и не будет ничего складывать.

При умножении линейной формы на число программа предлагает ввести число (вещественное, если коэффициенты линейной формы вещественные, или комплексное, если коэффициенты линейной формы комплексные), после чего умножает линейную форму на это число.

При вычислении значения программа предлагает пользователю ввести аргументы. После этого программа вычисляет значение линейной формы (вещественное при вещественных аргументах и комплексное при комплексных).

При выводе линейной формы программа просто выводит в консоль текущую линейную форму в формате $F(x) = a_0 + a_1*x_1 + a_2*x_2 + \ldots + a_n*x_n$.

При запуске тестов программа запускает UNIT-тестирование и выводит в консоль его результаты.

При выходе программа освобождает всю оставшуюся память и завершает свою работу.

Программный интерфейс

Реализован тип данных List. Этот тип данных представляет собой концепцию односвязного линейного списка.

Список методов класса List<T>:

Название метода	Аргументы	Описание
`int get_length();`		Возвращает значение length
`void push(T value);`	T value – значение, добавляемое в список	Добавляет элемент в конец линейной формы
`void pop();`		Удаляет элемент из конца линейной формы
`void print();`		Выводит линейную форму в консоль
`void sum_with(List<T>* sec_list);`	List<T>* sec_list – второй список	Складывает две линейные формы
`void multiply_by(T cnst);`	T cnst – константа, на которую умножается список	Умножает элементы линейной формы на число
`void correct_zeros();`		Корректирует линейную форму (удаляет все нулевые элементы, находящиеся в конце списка)
`T calculate_value(List<T> x)`	List<T> x – список аргументов линейной формы	Считает значение линейной формы по заданным аргументам

Тестирование

Тестирование разбито на 2 раздела - тестирование класса комплексных (3 модуля) и класса списка (6 модулей).

Файл с реализвацией тестирования `tests.hpp` находится в папке /Tests, а модули тестирования - в папках /Tests/Complex Tests/ и /Tests/List Tests/.

Для запуска тестирования нужно вызвать метод run_all_tests() или выбрать соответствующую команду в меню.

Список модулей тестирвания:

Раздел	Номер модуля	Файл модуля	Тестируемые методы
Complex	**1**	/Tests/Complex Tests/ complex_unit_1.cpp	`Complex(); Complex(double); Complex(double,double)`
	2	/Tests/Complex Tests/ complex_unit_2.cpp	`Перегрузка операторов сложения и произведения`
	3	/Tests/Complex Tests/ complex_unit_3.cpp	`Перегрузка операторов сравнения`
List	**1**	/Tests/List Tests/ list_unit_1.cpp	`Конструкторы списка`
	2	/Tests/List Tests/ list_unit_2.cpp	`push(T)`
	3	/Tests/List Tests/ list_unit_3.cpp	`pop()`
	4	/Tests/List Tests/ list_unit_4.cpp	`sum_with(List<T>*)`
	5	/Tests/List Tests/ list_unit_5.cpp	`multiply_by(T)`
	6	/Tests/List Tests/ list_unit_6.cpp	`calculate_value(List<T>)`

Приложение

main.cpp:

```cpp
#include "UI.hpp"
#include "list.hpp"
#include "complex.hpp"
#include "Tests/tests.hpp"
using namespace std;
int main() {
//    run_all_tests();
    run_UI();
    return 0;
}
```

UI.cpp:

```cpp
#include "UI.hpp"
using namespace std;
void run_UI(){
List<double> db_lform;
List<Complex> com_lform;
int dim = 0;
int type = 0;
int menu_item = 0;
while (menu_item != 7){
cout<<"\nPress\n";
cout<<" 1. to create new linear
form\n";
cout<<" 2. to add form with another
form\n";
```

```
cout<<" 3. to multiply by the
number\n";
cout<<" 4. to calculate value\n";
cout<<" 5. to show current linear
form\n";
cout<<" 6. to run tests\n";
cout<<" 7. to exit\n";
cin>>menu_item;
switch (menu_item) {
case 1:     //создание линейной формы
{
cout<<endl<<"Select type of linear
form:\n";
cout<<" 1. Real\n";
cout<<" 2. Complex\n";
                cin>>type;

                while (type != DOUBLE
&& type != COMPLEX){

cout<<endl<<"Incorrect type."<<endl;

cout<<endl<<"Select type of linear
form:\n";
                    cout<<" 1.
Real\n";
                    cout<<" 2.
Complex\n";
                    cin>>type;
                }

                if (type == DOUBLE) {
                    db_lform =
List<double>();
                }
                else {
                    com_lform =
List<Complex>();
                }
                cout<<endl<<"Enter
number of dimention:\n";
                cin>>dim;

                while (dim < 0) {

cout<<endl<<"Incorrect number."<<endl;
                    cout<<endl<<"Enter
number of dimention:\n";
                    cin>>dim;
                }
                if (type == DOUBLE){
                    for (int i = 0; i
<= dim; i++) {
                        int a = 0;
                        cout<<"\nEnter
a"<<i<<":\n";
                        cin>>a;

db_lform.push(a);
                    }
                    db_lform.print();
                }

                if (type == COMPLEX){
                    for (int i = 0; i
<= dim; i++) {
                        double a_re =
0;
                        double a_im =
0;
                        cout<<"\nEnter
real part of a"<<i<<":\n";
                        cin>>a_re;
                        cout<<"\nEnter
imaginary part of a"<<i<<":\n";
                        cin>>a_im;

com_lform.push(Complex(a_re, a_im));
                    }
                    com_lform.print();
                }

            }
                break;

            case 2:     //сложение
двух динейных форм
            {
                if (type == 0) {
                    cout<<"\nThe first
linear form doesn't exist.\n\n";
                    break;
                }

                if (type == DOUBLE) {
                    List<double>
sec_lform = List<double>();
                    int sec_dim = 0;
                    cout<<endl<<"Enter
number of dimention of the second
linear form:\n";
                    cin>>sec_dim;

                    while (sec_dim <
0) {

cout<<endl<<"Incorrect number."<<endl;

cout<<endl<<"Enter number of
dimention:\n";
                        cin>>sec_dim;
                    }

                    for (int i = 0; i
<= dim; i++) {
                        int a = 0;
                        cout<<"\nEnter
a"<<i<<":\n";
                        cin>>a;

sec_lform.push(a);
                    }


db_lform.sum_with(&sec_lform);
                    db_lform.print();
                }

                if (type == COMPLEX){
                    List<Complex>
sec_lform = List<Complex>();
                    int sec_dim = 0;
```

```
                    cout<<endl<<"Enter
number of dimention of the second
linear form:\n";
                    cin>>sec_dim;

                    while (sec_dim <
0) {

cout<<endl<<"Incorrect number."<<endl;

cout<<endl<<"Enter number of
dimention:\n";
                        cin>>sec_dim;
                    }

                    for (int i = 0; i
<= dim; i++) {
                        double a_re =
0;
                        double a_im =
0;
                        cout<<"\nEnter
real part of a"<<i<<":\n";
                        cin>>a_re;
                        cout<<"\nEnter
imaginary part of a"<<i<<":\n";
                        cin>>a_im;

sec_lform.push(Complex(a_re, a_im));
                    }


com_lform.sum_with(&sec_lform);
                    com_lform.print();
                }
            }
                break;

            case 3:     //умножение
линейной формы на константу
            {
                if (type == 0) {
                    cout<<"\nThe first
linear form doesn't exist.\n\n";
                    break;
                }

                if (type == DOUBLE) {
                    double cnst = 1;
                    cout<<"\nEnter
real constant:\n";
                    cin>>cnst;

db_lform.multiply_by(cnst);
                    db_lform.print();
                }

                if (type == COMPLEX){
                    double cnst_re =
1;
                    double cnst_im =
0;
                    cout<<"\nEnter
complex constant:\n";
                    cout<<" Enter the
real part:\n";
                    cin>>cnst_re;
                    cout<<" Enter the
imaginary part:\n";
                    cout<<cnst_im;

com_lform.multiply_by(Complex(cnst_re,
cnst_im));
                    com_lform.print();
                }
            }
                break;

            case 4:     //вычисление
значения линейной формы при известных
аргументах
            {
                if (type == 0) {
                    cout<<"\nThe first
linear form doesn't exist.\n\n";
                    break;
                }

                if (type == DOUBLE){
                    double val = 0;
                    List<double>
x_list = List<double>();
                    for (int i = 1; i
<= dim; i++){
                        double x = 0;
                        cout<<"\nEnter
x"<<i<<":\n";
                        cin>>x;

x_list.push(x);
                    }

                    val =
db_lform.calculate_value(x_list);
                    cout<<"Value =
"<<val;
                }

                if (type == COMPLEX){
                    Complex val = 0;
                    List<Complex>
x_list = List<Complex>();
                    for (int i = 1; i
<= dim; i++){
                        double x_re =
0;
                        double x_im =
0;
                        cout<<"\nEnter
real part of x"<<i<<":\n";
                        cin>>x_re;
                        cout<<"\nEnter
imaginary part of x"<<i<<":\n";
                        cin>>x_im;

x_list.push(Complex(x_re, x_im));
                    }

                    val =
com_lform.calculate_value(x_list);
                    cout<<"Value =
"<<val;
```

```
                }
            }
                break;

            case 5:     //вывести
текущую линейну форму в консоль
            {
                if (type == DOUBLE)
db_lform.print();
                if (type == COMPLEX)
com_lform.print();
            }
                break;

            case 6:     //запустить
тесты
            {
                run_all_tests();
            }
                break;

            case 7:     //выйти их
программы
                break;

            default:
                cout<<"Invalid
command\n\n";
                break;
        }
    }
}
```

UI.hpp:

```
#ifndef UI_hpp
#define UI_hpp

#define DOUBLE 1
#define COMPLEX 2

#include <stdio.h>
#include <iostream>
#include "list.hpp"
#include "complex.hpp"
#include "Tests/tests.hpp"

using namespace std;

void run_UI();

#endif /* UI_hpp */
```

list.cpp:

```
#include "list.hpp"

using namespace std;

//string double::to_string(){
//    std::ostringstream oss;
//    oss<<(*this);
//    return oss.str();
//}
```

```
template <typename T>
List<T>::List() {
    length = 0;
}

template <typename T>
List<T>::~List() {
//    while (head != NULL) {
//        ListItem<T>* current_item =
head->next;
//        delete head;
//        head = current_item;
//    }

    while (length != 0) pop();
}

template <typename T>       //
конструктор копирования
List<T>::List(const List &lst){
    length = 0;
    ListItem<T>* ptr_item = lst.head;

    if (lst.length != 0){
        while (ptr_item != 0){
            this->push(ptr_item-
>item);
            ptr_item = ptr_item->next;
        }
    }
}

//
// Нахождение знчения элемента списка
//
template <typename T>
T ListItem<T>::get_item(){
    return item;
}

//
// Нахождение длины списка
//
template <typename T>
int List<T>::get_length(){
    return length;
}

//
// Удаление последнего элемента в
списке
//
template <typename T>
void List<T>::pop() {
    if (length < 1) {
        throw out_of_range("Out of
range");
    }
    else{
        ListItem<T>* tmp_item =  this-
>head;
//        if (length != 1){
            for (int i = 0; i < length
- 1; i++){
```

```
                tmp_item = tmp_item->next;
            }
//        }

        delete tmp_item;
        length--;
    }
}

//
// Добаление элемента в конец списка
//
template <typename T>
void List<T>::push(T value) {
    ListItem<T>* new_item = new ListItem<T>();
    new_item->item = value;

    if (length == 0){
        head = new_item;
    }
    else{
        ListItem<T>* currtent_item = head;
        for (int i = 1; i < length; i++){
            currtent_item = currtent_item->next;
        }
        currtent_item->next = new_item;
    }

    length++;
}

//
// Вывод списка в консоль
//
template <typename T>
void List<T>::print(){
    cout<<"\nCurrent Linear form: \nF(x) = ";
    bool is_form_null = true;       // равна ли линейная форма нулю

    if(this->get_length() > 0){
        for (int i = 0; i < this->get_length(); i++){   //проверка на существование ненулевых элементов в линейной форме
            if ((*this)[i] != 0){
                is_form_null = false;
                break;
            }
        }
    }
    if (is_form_null) cout<<"0";
//если в линейной форме все коэфициенты равны 0, то выводим ноль

    else{
        for (int i = 0; i < this->get_length(); i++){
            if ((*this)[i] != 0){
                if (i == 0)
                    cout<<(*this)[i]<<" ";
                else
                    cout<<"+("<<(*this)[i]<<")x"<<i<<" ";
            }
        }
    }
    cout<<endl;
}

//
// Взятие значения из списка по индексу
//
template <typename T>
T List<T>::operator[](int index){
    if ((0 > index) || (index >= length )) {
        throw std::out_of_range("Out of range");
    }
    else{
        ListItem<T>* tmp_item = head;

        for (int i = 0; i < index; i++){
            tmp_item = tmp_item->next;
        }

        return tmp_item->get_item();
    }
}

//
// Сложение двух списков
//
template <typename T>
void List<T>::sum_with(List<T>* sec_list){

    if (this->length == sec_list->length){
        ListItem<T>* current_item_1 = this->head;
        ListItem<T>* current_item_2 = sec_list->head;

        for (int i = 0; i < this->length; i++){
            current_item_1->item = current_item_1->item + current_item_2->item;
            current_item_1 = current_item_1->next;
            current_item_2 = current_item_2->next;
        }
    }
    else if (this->length > sec_list->length){
        ListItem<T>* current_item_1 = this->head;
        ListItem<T>* current_item_2 = sec_list->head;
```

```
        for (int i = 0; i < sec_list->length; i++){
            current_item_1->item = current_item_1->item + current_item_2->item;
            current_item_1 = current_item_1->next;
            current_item_2 = current_item_2->next;
        }
    }
    else {
        ListItem<T>* current_item_1 = this->head;
        ListItem<T>* current_item_2 = sec_list->head;

        for (int i = 0; i < this->length; i++){
            current_item_1->item = current_item_1->item + current_item_2->item;
            current_item_1 = current_item_1->next;
            current_item_2 = current_item_2->next;
        }
        for (int i = this->length; i < sec_list->length; i++){

(*this).push(current_item_2->item);
            current_item_2 = current_item_2->next;
        }
    }

    (*this).correct_zeros();
}

//
// Умножение списка на число
//
template <typename T>
void List<T>::multiply_by(T cnst){
    ListItem<T>* current_item = this->head;

    for (int i = 0; i < this->length; i++){
        current_item->item = current_item->item * cnst;
        current_item = current_item->next;
    }
    (*this).correct_zeros();
}

//
// Проверка на наличие в конце списка нулей и корректирование списка
//
template <typename T>
void List<T>::correct_zeros(){
    while (this->length > 0 && (*this)[this->length - 1] == 0){
        (*this).pop();
    }
}

template <typename T>
T List<T>::calculate_value(List<T> x){
    if (x.length == 0 || this->length == 0) return 0;
    T val = 0;
    val += this->head->item;
    ListItem<T>* current_item = this->head;
    for (int i = 0; i < this->length - 1; i++){
        current_item = current_item->next;
        val += current_item->item * x[i];
    }
    return val;
}
```

list.hpp:

```
#ifndef list_hpp
#define list_hpp

#include <stdio.h>
#include <stdexcept>
#include <iostream>
#include <string>
#include <sstream>

using namespace std;

template <typename T>
struct ListItem {
    T item;
    ListItem* next;

    T get_item();
};

template <typename T>
class List{
private:
    ListItem<T>* head;
    int length;

public:
    List();
    ~List();
    List(const List &lst); // конструктор копирования

    int get_length();
    void push(T value);
    void pop();
    void print();
    void sum_with(List<T>* sec_list);
    void multiply_by(T cnst);
    void correct_zeros();

    T operator[](int index);
```

```
//    List<T> operator+ (List<T>
sec_list);
    T calculate_value(List<T> x);
};

#include "list.cpp"

#endif /* list_hpp */
```

complex.cpp:

```
#include "complex.hpp"

using namespace std;

Complex::Complex(double r, double i){
    re = r;
    im = i;
}

Complex::Complex(double r){
    re = r;
    im = 0;
}

Complex::Complex(){
    re = 0;
    im = 0;
}

Complex::~Complex(){}

double Complex::return_re(){
    return this->re;
}

double Complex::return_im(){
    return this->im;
}

Complex Complex::operator+ (Complex
com){
    return Complex(this->re + com.re,
this->im + com.im);
}

Complex Complex::operator+= (Complex
com){
    return Complex(this->re = this->re
+ com.re,this->im = this->im +
com.im);
}

Complex Complex::operator*(Complex
com){
    return Complex(this->re * com.re -
this->im * com.im, this->re * com.im +
this->im * com.re);
}

Complex Complex::operator+ (double db)
{
    return Complex(this->re + db,
this->im);
}

Complex Complex::operator+= (double
db){
    return Complex(this->re = this->re
+ db, this->im);
}

Complex Complex::operator* (double db)
{
    return Complex(this->re * db,
this->im * db);
}

Complex Complex::operator*= (double
db){
    return Complex(this->re = this->re
* db,this->im =  this->im * db);
}

bool Complex::operator==(Complex com){
    return this->re == com.re && this-
>im == com.im;
}

bool Complex::operator!=(Complex com){
    return this->re != com.re || this-
>im != com.im;
}

bool Complex::operator==(double db){
    return this->re == db && this->im
== 0;
}

bool Complex::operator!=(double db){
    return this->re != db || this-
>im != 0;
}

void Complex::print(){
    if (this->im >= 0)  cout<<this-
>re<<" + "<<this->im<<"i\n";
    else cout<<this->re<<" - "<<-
(this->im)<<"i\n";
}

std::ostream & operator<<(std::ostream
& os, const Complex & c)
{
    if (c.im == 0) os<<c.re<<" ";
    else os<<c.re<<(c.im > 0 ? "+" :
"-")<<abs(c.im)<<"i";
    return os;
}
```

complex.hpp:

```
#ifndef complex_hpp
#define complex_hpp

#include <stdio.h>
```

```
#include <stdlib.h>
#include <iostream>
#include <string>
#include <sstream>
#include <cmath>

using namespace std;

class Complex{
private:
    double re, im;

public:
    Complex (double, double);
    Complex (double);
    Complex();
    ~Complex();

    double return_re();
    double return_im();

    Complex operator+ (Complex);
    Complex operator+= (Complex);
    Complex operator* (Complex);
    Complex operator+ (double);
    Complex operator* (double);
    Complex operator+= (double);
    Complex operator*= (double);
    bool operator== (Complex);
    bool operator!= (Complex);
    bool operator== (double);
    bool operator!= (double);

    void print();

    friend std::ostream &
operator<<(std::ostream &os, const
Complex &c);
};

#endif /* complex_hpp */
```

Tests/tests.cpp:

```
#include "tests.hpp"

void test_prepare(int *test_id, bool
*test_ok){
    (*test_ok) = false;
    (*test_id)++;

    cout<<"[TESTING] Test
#"<<*test_id<<": ";
}

void test_result(bool test_ok){
    if (test_ok == true){
        cout<<"OK.\n";
    } else {
        cout<<"ERROR.\n";
    }
}

void run_all_tests(){
    run_complex_tests();
    run_list_tests();
}


void run_complex_tests(){
    run_complex_unit_1();
    run_complex_unit_2();
    run_complex_unit_3();
}

void run_list_tests(){
    run_list_unit_1();
    run_list_unit_2();
    run_list_unit_3();
    run_list_unit_4();
    run_list_unit_5();
    run_list_unit_6();
}
```

Tests/tests.hpp:

```
#ifndef tests_hpp
#define tests_hpp

#include <stdio.h>
#include <iostream>
#include "Complex Tests/
complex_unit_1.hpp"
#include "Complex Tests/
complex_unit_2.hpp"
#include "Complex Tests/
complex_unit_3.hpp"
#include "List Tests/list_unit_1.hpp"
#include "List Tests/list_unit_2.hpp"
#include "List Tests/list_unit_3.hpp"
#include "List Tests/list_unit_4.hpp"
#include "List Tests/list_unit_5.hpp"
#include "List Tests/list_unit_6.hpp"

using namespace std;

void run_all_tests();
void run_complex_tests();
void run_linearForm_test();
void run_list_tests();
void test_prepare(int *test_id, bool
*test_ok);
void test_result(bool test_ok);

#endif /* tests_hpp */
```

Tests/List Tests/list_unit_1.cpp:

```
#include "list_unit_1.hpp"

/*
 UNIT 1: Тестирование создания списка

 Список тестов:
 Тест 1: Создание пустого списка
 */
```

```
void run_list_unit_1(){
    cout<<"[LIST CLASS TESTING] UNIT 1
:\n\n";
    int test_id = 0;
    bool test_ok = true;

    //TECT 1
    if (test_ok == true){
        test_prepare(&test_id,
&test_ok);

        List<int> lst;
        if (lst.get_length() == 0)
test_ok = true;
        else test_ok = false;

        test_result(test_ok);

    }

    if (test_id == 1 && test_ok ==
true) {
        cout<<"\n[TESTING] Unit 1
testing SUCCESSEDED.\n\n\n";
    } else {
        cout<<"\n[TESTING] Unit 1
testing FAILED on TEST #"<<test_id<<".
\n\n\n";
    }
}
```

Tests/List Tests/list_unit_1.hpp:

```
#ifndef list_unit_1_hpp
#define list_unit_1_hpp

#include <stdio.h>
#include <iostream>
#include "tests.hpp"
#include "list.hpp"

void run_list_unit_1();

#endif /* list_unit_1_hpp */
```

Tests/List Tests/list_unit_2.cpp:

```
#include "list_unit_2.hpp"

/*
 UNIT 2: Тестирование добавления
элемента в список

 Список тестов:
 Тест 1: добавление элемента в пустой
список
 Тест 2: добавление элемента в
непустой список
 */

void run_list_unit_2(){
    cout<<"[LIST CLASS TESTING] UNIT 2
:\n\n";
    int test_id = 0;
    bool test_ok = true;

    //TECT 1
    if (test_ok == true){
        test_prepare(&test_id,
&test_ok);

        List<int> lst;
        lst.push(4);
        if (lst[0] == 4 &&
lst.get_length() == 1) test_ok = true;

        test_result(test_ok);
    }

    //TECT 2
    if (test_ok == true){
        test_prepare(&test_id,
&test_ok);

        List<Complex> lst;
        lst.push(Complex(2,5));
        lst.push(Complex(4,2));
        if (lst[1] == Complex(4,2) &&
lst.get_length() == 2) test_ok = true;

        test_result(test_ok);
    }

    if (test_id == 2 && test_ok ==
true) {
        cout<<"\n[TESTING] Unit 2
testing SUCCESSEDED.\n\n\n";
    } else {
        cout<<"\n[TESTING] Unit 2
testing FAILED on TEST #"<<test_id<<".
\n\n\n";
    }
}
```

Tests/List Tests/list_unit_2.hpp:

```
#ifndef list_unit_2_hpp
#define list_unit_2_hpp

#include <stdio.h>
#include <iostream>
#include "tests.hpp"
#include "list.hpp"
#include "complex.hpp"

void run_list_unit_2();

#endif /* list_unit_2_hpp */
```

Tests/List Tests/list_unit_3.cpp:

```
#include "list_unit_3.hpp"

/*
```

```
 UNIT 3: Тестирование удаления
элемента из списока

 Список тестов:
 Тест 1: удаление элемента из списка
 Тест 2: Удаление нескольких элементов
из списка
 Тест 3: Удаление элемента из пустого
списка
 */

void run_list_unit_3(){
    cout<<"[LIST CLASS TESTING] UNIT 3
:\n\n";
    int test_id = 0;
    bool test_ok = true;

    //TECT 1
    if (test_ok == true){
        test_prepare(&test_id,
&test_ok);

        List<int> lst;
        lst.push(4);
        lst.pop();
        if (lst.get_length() == 0)
test_ok = true;

        test_result(test_ok);
    }

    //TECT 2
    if (test_ok == true){
        test_prepare(&test_id,
&test_ok);

        List<Complex> lst;
        lst.push(Complex(2,5));
        lst.push(Complex(4,2));
        lst.pop();
        if (lst[0] == Complex(2,5) &&
lst.get_length() == 1) test_ok = true;

        test_result(test_ok);
    }

    //TECT 3
    if (test_ok == true){
        test_prepare(&test_id,
&test_ok);

        List<double> lst;
        try {
            lst.pop();
        } catch (std::out_of_range
exep) {
            test_ok = true;
        }

        if (lst.get_length() != 0)
test_ok = false;

        test_result(test_ok);
    }

    if (test_id == 3 && test_ok ==
true) {
        cout<<"\n[TESTING] Unit 3
testing SUCCESSEDED.\n\n\n";
    } else {
        cout<<"\n[TESTING] Unit 3
testing FAILED on TEST #"<<test_id<<".
\n\n\n";
    }
}
```

Tests/List Tests/list_unit_3.hpp:

```
#ifndef list_unit_3_hpp
#define list_unit_3_hpp

#include <stdio.h>
#include <iostream>
#include "tests.hpp"
#include "list.hpp"

void run_list_unit_3();

#endif /* list_unit_3_hpp */
```

Tests/List Tests/list_unit_4.cpp:

```
#include "list_unit_4.hpp"

/*
 UNIT 4: Тестирование сложения списков

 Список тестов:
 Тест 1: Сложение двух нулевых списков
 Тест 2: Сложение нулевого списка и
ненулевого
 Тест 3: Сложение ненулевого списка и
нулевого
 Тест 4: Сложение двух ненулевых
списков разной размерности
 Тест 5: Сложение противоположных по
знаку списков
 */

void run_list_unit_4(){
    cout<<"[LIST CLASS TESTING] UNIT 4
:\n\n";
    int test_id = 0;
    bool test_ok = true;

    //TECT 1
    if (test_ok == true){
        test_prepare(&test_id,
&test_ok);

        List<int> lst1;
        List<int> lst2;

        lst1.sum_with(&lst2);

        if (lst1.get_length() == 0)
test_ok = true;
```

```
        test_result(test_ok);
    }

    //TECT 1
    if (test_ok == true){
        test_prepare(&test_id,
&test_ok);

        List<double> lst1;
        List<double> lst2;
        lst2.push(5.6);
        lst2.push(3.1);

        lst1.sum_with(&lst2);

        if (lst1.get_length() == 2)
            if (lst1[0] == 5.6)
                if (lst1[1] == 3.1)
test_ok = true;

        test_result(test_ok);
    }

    if (test_ok == true){
        test_prepare(&test_id,
&test_ok);

        List<Complex> lst1;
        List<Complex> lst2;
        lst1.push(Complex(5, 6));
        lst1.push(Complex(3, 1));

        lst1.sum_with(&lst2);

        if (lst1.get_length() == 2)
            if (lst1[0] == Complex(5,
6))
                if (lst1[1] ==
Complex(3, 1)) test_ok = true;

        test_result(test_ok);
    }

    //    TECT 4
    if (test_ok == true){
        test_prepare(&test_id,
&test_ok);

        List<double> lst1;
        lst1.push(1);
        lst1.push(2);
        lst1.push(3);
        lst1.push(4);
        lst1.push(5);

        List<double> lst2;
        lst2.push(2);
        lst2.push(3);

        lst1.sum_with(&lst2);

        if (lst1.get_length() == 5)
            if (lst1[0] == 3)
                if (lst1[1] == 5)
                    if (lst1[2] == 3)
                        if (lst1[3] ==
4)
                            if
(lst1[4] == 5) test_ok = true;

        test_result(test_ok);
    }

    //    TECT 4
    if (test_ok == true){
        test_prepare(&test_id,
&test_ok);

        List<double> lst1;
        lst1.push(1);
        lst1.push(2);
        lst1.push(3);
        lst1.push(4);
        lst1.push(5);

        List<double> lst2;
        lst2.push(-1);
        lst2.push(-2);
        lst2.push(-3);
        lst2.push(-4);
        lst2.push(-5);

        lst1.sum_with(&lst2);

        if (lst1.get_length() == 0)
test_ok = true;

        test_result(test_ok);
    }

    if (test_id == 5 && test_ok ==
true) {
        cout<<"\n[TESTING] Unit 4
testing SUCCESSEDED.\n\n\n";
    } else {
        cout<<"\n[TESTING] Unit 4
testing FAILED on TEST #"<<test_id<<".
\n\n\n";
    }
}
```

Tests/List Tests/list_unit_4.hpp:

```
#ifndef list_unit_4_hpp
#define list_unit_4_hpp

#include <stdio.h>
#include <iostream>
#include "tests.hpp"
#include "list.hpp"
#include "complex.hpp"

void run_list_unit_4();

#endif /* list_unit_4_hpp */
```

Tests/List Tests/list_unit_5.cpp:

```
#include "list_unit_5.hpp"

/*
 UNIT 5: Тестирование умножения списка
на число

 Список тестов:
 Тест 1: Умножение списка на ненулевую
величину
 Тест 2: Умножение списка на 0
 */

void run_list_unit_5(){
    cout<<"[LIST CLASS TESTING] UNIT 5
:\n\n";
    int test_id = 0;
    bool test_ok = true;

    //TECT 1
    if (test_ok == true){
        test_prepare(&test_id,
&test_ok);

        List<double> lst;
        lst.push(1);
        lst.push(2);
        lst.push(3);
        lst.push(4);
        lst.push(5);

        lst.multiply_by(5);

        if (lst.get_length() == 5)
            if (lst[0] == 5)
                if (lst[1] == 10)
                    if (lst[2] == 15)
                        if (lst[3] ==
20)
                            if (lst[4]
== 25) test_ok = true;

        test_result(test_ok);
    }

    //TECT 2
    if (test_ok == true){
        test_prepare(&test_id,
&test_ok);

        List<double> lst;
        lst.push(1);
        lst.push(2);
        lst.push(3);
        lst.push(4);
        lst.push(5);

        lst.multiply_by(0);

        if (lst.get_length() == 0)
test_ok = true;

        test_result(test_ok);
    }

    if (test_id == 2 && test_ok ==
true) {
      cout<<"\n[TESTING] Unit 5
testing SUCCESSEDED.\n";

cout<<"-------------------------------
---------"<<"\n\n\n";
    } else {
    cout<<"\n[TESTING] Unit 5 testing
FAILED on TEST #"<<test_id<<".\n";

cout<<"-------------------------------
---------"<<"\n\n\n";
    }
}
```

Tests/List Tests/list_unit_5.hpp:

```
#ifndef list_unit_5_hpp
#define list_unit_5_hpp

#include <stdio.h>
#include <iostream>
#include "tests.hpp"
#include "list.hpp"

void run_list_unit_5();

#endif /* list_unit_5_hpp */
```

Tests/List Tests/list_unit_6.cpp:

```
#include "list_unit_6.hpp"

/*
 UNIT 6: Тестирование вычисления
значения при заданных значениях
аргументов

 Список тестов:
 Тест 1: нахождение значения при
нулвеых аргументах
 Тест 2: нахождение значения при
нудевых коэфициентах
 Тест 3: нахождение значения
 */

void run_list_unit_6(){
    cout<<"[LIST CLASS TESTING] UNIT 6
:\n\n";
    int test_id = 0;
    bool test_ok = true;

    //TECT 1
    if (test_ok == true){
        test_prepare(&test_id,
&test_ok);

        List<double> lst;
        lst.push(1);
        lst.push(2);
        lst.push(3);
        lst.push(4);
```

```
        lst.push(5);

        List<double> x;
        double val =
lst.calculate_value(x);
        if (val == 0) test_ok = 1;

        test_result(test_ok);
    }

    //TECT 2
    if (test_ok == true){
        test_prepare(&test_id,
&test_ok);

        List<double> lst;

        List<double> x;
        x.push(1);
        x.push(2);
        x.push(3);
        x.push(4);
        x.push(5);

        double val =
lst.calculate_value(x);

        if (val == 0) test_ok = 1;

        test_result(test_ok);
    }

    //TECT 3
    if (test_ok == true){
        test_prepare(&test_id,
&test_ok);

        List<double> lst;
        lst.push(1);
        lst.push(2);
        lst.push(3);
        lst.push(4);
        lst.push(5);

        List<double> x;
        x.push(5);
        x.push(4);
        x.push(3);
        x.push(2);

        double val =
lst.calculate_value(x);

        if (val == 45) test_ok = 1;

        test_result(test_ok);
    }

    if (test_id == 3 && test_ok ==
true) {
        cout<<"\n[TESTING] Unit 6
testing SUCCESSEDED.\n";

cout<<"------------------------------
---------"<<"\n\n\n";
    } else {
        cout<<"\n[TESTING] Unit 6
testing FAILED on TEST #"<<test_id<<".
\n";

cout<<"------------------------------
---------"<<"\n\n\n";
    }
}
```

Tests/List Tests/list_unit_6.hpp:

```
#ifndef list_unit_6_hpp
#define list_unit_6_hpp

#include <stdio.h>
#include <iostream>
#include "tests.hpp"
#include "list.hpp"
#include "complex.hpp"

void run_list_unit_6();

#endif /* list_unit_6_hpp */
```

Tests/Complex Tests/complex_unit_1.cpp:

```
#include "complex_unit_1.hpp"

/*
 UNIT 1: Тестирование создания
комплексного числа

 Список тестов:
 Тест 1: Создание комплексного числа с
нулевой действительной частью и
нулевой мнимой частью
 Тест 2: Создание комплексного числа с
ненулевой действительной частью и
нулевой мнимой частью
 Тест 2: Создание комплексного числа с
ненулевой действительной частью и
ненулевой мнимой частью
 */

void run_complex_unit_1(){
    cout<<"[COMPLEX CLASS TESTING]
UNIT 1 :\n\n";
    int test_id = 0;
    bool test_ok = true;

    //TECT 1
    if (test_ok == true){
        test_prepare(&test_id,
&test_ok);

        Complex com = Complex();


        if (com.return_re() == 0 &&
com.return_im() == 0){
            test_ok = true;
        }
```

```
        test_result(test_ok);
    }

    //ТЕСТ 2
    if (test_ok == true){
        test_prepare(&test_id,
&test_ok);

        Complex com = Complex(5);


        if (com.return_re() == 5 &&
com.return_im() == 0){
            test_ok = true;
        }

        test_result(test_ok);
    }

    //ТЕСТ 3
    if (test_ok == true){
        test_prepare(&test_id,
&test_ok);

        Complex com = Complex(5, -2);


        if (com.return_re() == 5 &&
com.return_im() == -2){
            test_ok = true;
        }

        test_result(test_ok);
    }

    if (test_id == 3 && test_ok ==
true) {
        cout<<"\n[TESTING] Unit 1
testing SUCCESSEDED.\n\n\n";
    } else {
        cout<<"\n[TESTING] Unit 1
testing FAILED on TEST #"<<test_id<<".
\n\n\n";
    }
}
```

Tests/Complex Tests/complex_unit_1.hpp:

```
#ifndef complex_unit_1_hpp
#define complex_unit_1_hpp

#include <stdio.h>
#include "complex.hpp"
#include "tests.hpp"

void run_complex_unit_1(void);

#endif /* complex_unit_1_hpp */
```

Tests/Complex Tests/complex_unit_2.cpp:

```
#include "complex_unit_2.hpp"

/*
 UNIT 2: Тестирование математических
операций с классом Complex

 Список тестов:
 Тест 1: Сложение комплексного и
комплексного (операции + и +=)
 Тест 2: Сложение комплексного и
вещественного (операции + и +=)
 Тест 3: Произведение комплексного и
вещественного (операции * и *=)
 Тест 4: Произведение комплексного и
комплексного
 */

void run_complex_unit_2(){
    cout<<"[COMPLEX CLASS TESTING]
UNIT 2 :\n\n";
    int test_id = 0;
    bool test_ok = true;

    //ТЕСТ 1
    if (test_ok == true){
        test_prepare(&test_id,
&test_ok);

        Complex com1 = Complex(2, 5);
        Complex com2 = Complex(3, 4);

        Complex test_com_1 = com1 +
com2;
        Complex test_com_2 = com1;
        test_com_2 += com2;

        if (test_com_1.return_re() ==
5 && test_com_1.return_im() == 9 &&
test_com_2.return_re() == 5 &&
test_com_2.return_im() == 9){
            test_ok = true;
        }

        test_result(test_ok);
    }

    //ТЕСТ 2
    if (test_ok == true){
        test_prepare(&test_id,
&test_ok);

        Complex com1 = Complex(2, 5);
        double db = 5;

        Complex test_com_1 = com1 +
db;
        Complex test_com_2 = com1;
        test_com_2 += db;

        if (test_com_1.return_re() ==
7 && test_com_1.return_im() == 5 &&
test_com_2.return_re() == 7 &&
test_com_2.return_im() == 5){
            test_ok = true;
        }

        test_result(test_ok);
```

```
    }

    //ТЕСТ 3
    if (test_ok == true){
        test_prepare(&test_id,
&test_ok);

        Complex com1 = Complex(2, 5);
        double db = 5;

        Complex test_com_1 = com1 *
db;
        Complex test_com_2 = com1;
        test_com_2 *= db;

        if (test_com_1.return_re() ==
10 && test_com_1.return_im() == 25 &&
test_com_2.return_re() == 10 &&
test_com_2.return_im() == 25){
            test_ok = true;
        }

        test_result(test_ok);
    }

    //ТЕСТ 4
    if (test_ok == true){
        test_prepare(&test_id,
&test_ok);

        Complex com1 = Complex(2, 5);
        Complex com2 = Complex(3, 4);

        Complex test_com = com1 *
com2;

        if (test_com == Complex(-14,
23)){
            test_ok = true;
        }

        test_result(test_ok);
    }

    if (test_id == 4 && test_ok ==
true) {
        cout<<"\n[TESTING] Unit 2
testing SUCCESSEDED.\n\n\n";
    } else {
        cout<<"\n[TESTING] Unit 2
testing FAILED on TEST #"<<test_id<<".
\n\n\n";
    }
}
```

Tests/Complex Tests/complex_unit_2.hpp:

```
#ifndef complex_unit_2_hpp
#define complex_unit_2_hpp

#include <stdio.h>
#include "complex.hpp"
#include "tests.hpp"

void run_complex_unit_2(void);

#endif /* complex_unit_2_hpp */
```

Tests/Complex Tests/complex_unit_3.cpp:

```
#include "complex_unit_3.hpp"

/*
 UNIT 3: Тестирование операций
сравнения

 Список тестов:
 Тест 1: Сравнение двух совпадающих
комплексных (для == и !=)
 Тест 2: Сравнение двух несовпадающих
комплексных (для == и !=)
 Тест 3: Сравнение совпадающих
комплексного и вещественного (для == и
!=)
 Тест 4: Сравнение несовпадающих
комплексного и вещественного (для == и
!=)
 */

void run_complex_unit_3(){
    cout<<"[COMPLEX CLASS TESTING]
UNIT 3 :\n\n";
    int test_id = 0;
    bool test_ok = true;

    //ТЕСТ 1
    if (test_ok == true){
        test_prepare(&test_id,
&test_ok);

        Complex com1 = Complex(3, 4);
        Complex com2 = Complex(3, 4);

        bool test1_ok = (com1 ==
com2);
        bool test2_ok = (com1 !=
com2);

        if (test1_ok && !test2_ok){
            test_ok = true;
        }

        test_result(test_ok);
    }

    //ТЕСТ 2
    if (test_ok == true){
        test_prepare(&test_id,
&test_ok);

        Complex com1 = Complex(3, 4);
        Complex com2 = Complex(5, -4);

        bool test1_ok = (com1 ==
com2);
        bool test2_ok = (com1 !=
com2);
```

```
        if (!test1_ok && test2_ok){
            test_ok = true;
        }

        test_result(test_ok);
    }

    //TECT 3
    if (test_ok == true){
        test_prepare(&test_id,
&test_ok);

        Complex com = Complex(3, 0);
        double db = 3;

        bool test1_ok = (com == db);
        bool test2_ok = (com != db);

        if (test1_ok && !test2_ok){
            test_ok = true;
        }

        test_result(test_ok);
    }

    //TECT 4
    if (test_ok == true){
        test_prepare(&test_id,
&test_ok);

        Complex com = Complex(3, 7);
        double db = 3;

        bool test1_ok = (com == db);
        bool test2_ok = (com != db);

        if (!test1_ok && test2_ok){
            test_ok = true;
        }

        test_result(test_ok);
    }

    if (test_id == 4 && test_ok ==
true) {
        cout<<"\n[TESTING] Unit 3
testing SUCCESSEDED.\n";

cout<<"------------------------------
---------"<<"\n\n\n";
    } else {
        cout<<"\n[TESTING] Unit 3
testing FAILED on TEST #"<<test_id<<".
\n";

cout<<"------------------------------
---------"<<"\n\n\n";
    }
}
```

Tests/Complex Tests/complex_unit_3.hpp:

```
#ifndef complex_unit_3_hpp
#define complex_unit_3_hpp

#include <stdio.h>
#include "complex.hpp"
#include "tests.hpp"

void run_complex_unit_3(void);

#endif /* complex_unit_3_hpp */
```