НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ЯДЕРНЫЙ УНИВЕРСИТЕТ «МИФИ»

ИИКС

Кафедра №22 «Кибернетика»

Лабораторная работа №3

Выполнил:

студент гр. Б17-514

Жарков М. А.

Преподаватель:

Трифоненков А.В.

Москва 2018

Вариант №9

Коллекция: Бинарное дерево поиска

Типы хранимых данных: вещественные числа, комплексные числа

Операции над коллекцией: map, where, слияние, извлечение поддерева по заданному элементу, поиск элемента на вхождение, сохранение в строку в соответствии с обходами ЛПК и ПЛК.

Бинарное дерево поиска - это двоичное дерево, для которого выполняются следующие дополнительные условия:
- Оба поддерева — левое и правое — являются двоичными деревьями поиска.
- У всех узлов левого поддерева произвольного узла X значения ключей данных меньше, нежели значение ключа данных самого узла X.
- У всех узлов правого поддерева произвольного узла X значения ключей данных больше либо равны, нежели значение ключа данных самого узла X.

Операция map и where: Если $l = [a_1, .., a_n]$ - некоторый список элементов типа T, а $f : T \to T$, то:

$\mathrm{map}(f, l) \mapsto \left[f(a_1), .., f(a_n)\right]$Если, при тех же соглашениях, $h : T \to \mathrm{Bool}$ - некоторая функция, возвращающая булево значение, то результатом $\mathrm{where}(h, l)$ будет новый список l', такой что: $a'_i \in l' \Leftrightarrow h(a'_i) = \mathrm{true}$. Т.е. where фильтрует значения из списка l с помощью функции-фильтра h.

Извлечение поддерева по заданному элементу - нахождение элемента в в дереве и удаление поддерева, корнем которого является данный элемент.

Поиск элемента на вхождение - булева операция, возвращающая истину, если элемент присутствует в дереве.

Сохранение в строку в соответствии с обходом - перевод дерева в формат строки в виде [Л][П]К или [П][Л]К.

Техническое задание

Создать заголовочный файл с обьявлением класса бинарного дерева **tree.hpp** и класса комплексных **complex.cpp**

Объявить шаблон класса Tree<T> со следующими полями и разработать реализацию методов этого шаблона:

Название поля	Описание
Private	
`Node<T>* Root`	`Указатель на корень дерева`
`void remove_subtree(Node<T> *)`	`вспомогательная функция для удаления поддерева`
`Node<T>* merge(Node<T>*, Tree<T> &);`	`вспомогательная функция для merge`
`void print(Node<T>*, int);`	`вспомогательная функция для print`

Название поля	Описание
`int get_depth(Node<T>*, int);`	вспомогательная функция для get_depth
Public	
`Tree();`	Конструктор
`~Tree();`	Деструктор
`void add_node(T);`	Добавление элемента в дерево
`void remove_node(T);`	удаление элемента из дерево по значению
`void remove_node(Node<T>*);`	удаление элемента из дерево по узлу
`void clear_node(Node<T>*);`	Обнуляет указатель
`string post_order_to_str(Node<T>* x);`	обход левый-правый-корень, начиная с x
`string post_order_to_str();`	обход левый-правый-корень, начиная с корня
`string reverse_post_order_to_str(Node<T>* x);`	обход правый-левый-корень, начиная с x
`string reverse_post_order_to_str();`	обход правый-левый-корень, начиная с корня
`Node<T>* find_node(T);`	находит узел по значению
`bool is_in_tree(T);`	проверка элемента на вхождение
`Node<T>* find_min_node(Node<T>*);`	Находит минимальное значение, начиная с узла
`Node<T>* find_max_node(Node<T>*);`	Находит максимальное значение, начиная с узла
`Node<T>* copy_node(Node<T>*, Tree<T>&);`	копирует узел
`Tree<T> copy_tree();`	копирует дерево
`Tree<T> copy_tree(Node<T>*);`	Вспомогательная функция для копирования дерева
`Node<T>* get_root();`	Возвращает корень
`void remove_subtree(T val);`	Удаляет поддерево
`typedef bool (*where_func)(T);`	Where функция
`Node<T>* where(where_func, Node<T>*, Tree<T>&);`	Вспомогательная функция для where
`Tree<T> where(where_func wf);`	Where функция
`Tree<T> merge(Tree<T> &sec_tree);`	прибавляет второе дерево к первому (слияние)
`typedef T (*map_func)(T);`	Map функция
`void  map(Node<T> *, map_func);`	вспомогательная функция для map
`void map(map_func mf);`	Map функция

Название поля	Описание
`void print();`	Вывод дерева в консоль
`int get_depth();`	находит глубину дерева

Так же объявляется шаблон структуры Node<T>:

Название поля	Описание
`T data;`	Значение узла
`Node<T>* left;`	Указатель на левого потомка
`Node<T>* right;`	Указатель на правого потомка
`Node<T>* parent;`	Указатель на родителя
`Node();`	Конструктор
`Node(const T);`	Деструктор

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
`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

Название поля	Описание
`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
`bool operator< (Complex);` `bool operator<= (Complex);`	Перегрузка операторов сравнения
`bool operator> (Complex);` `bool operator>= (Complex);`	Перегрузка операторов сравнения

Пользовательский интерфейс

В программе реализован консольный интерфейс, с помощью которого пользователь создает дерево и работает с ним. 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	Добавление элемента в дерево
2	Удаление элемента из дерева
3	Вывод дерева в консоль
4	Использование map функции
5	Использование where функции
6	Слияние двух деревьев
7	Извлечение поддерева
8	Поиск элемента на вхождение
9	Вывод в формате ЛПК
10	Вывод в формате ПЛК
11	Запуск тестов

Номер команды	Результат выполнение команды
12	Выход из программы

При добавлении элемента пользователь выбирает тип значения, с которым будет работать дерево (вещественные или комплексные), а далее вводит это значение.

При удалении элемента, пользователь вводит значение, и оно (если оно есть в дереве) удаляется.

Использование функции map возводит значения всех элементов дерева в квадрат.

Использование функции where удаляет все неположительные элементы.

При использовании merge программа предложит ввести второе дерево, после чего произведет слияние деревьев.

При использовании удаления поддерева программа попросит пользователя ввести значение, после чего удалит поддерево, корнем которого будет являться это значение.

При выводе обхода программа выведет дерево в формате [Левый потомок][Правый потомок]Корень или [Правый потомок][Левый потомок]Корень

Программный интерфейс

Реализован тип данных Tree. Этот тип данных представляет собой концепцию бинарного дерева поиска.

Список основных методов класса Tree<T>:

Название метода	Аргументы	Описание
`void add_node(T);`	Значение типа T	Добавление элемента в дерево
`void remove_node(T);`	Значение типа T	удаление элемента из дерево по значению
`string post_order_to_str();`		Выводит строку в соответствии с обходом ЛПК
`string reverse_post_order_to_str();`		Выводит строку в соответствии с обходом ПЛК
`Node<T>* find_node(T);`	Значение типа T	Находит узел со значением
`bool is_in_tree(T);`	Значение типа T	проверка элемента на вхождение
`void remove_subtree(T val);`	Значение типа T	Удалет поддерево
`Tree<T> where(where_func wf);`	wf – функция where	Where функция
`void map(map_func mf);`	mp – функция map	Map функция

Название метода	Аргументы	Описание
`void print();`		Выводит дерево в консоль
`int get_depth();`		Возвращает глубину дерева

Тестирование

Тестирование разбито на 2 раздела - тестирование класса комплексных (3 модуля) и класса дерева (5 модулей).

Файл с реализвацией тестирования `tests.hpp` находится в папке /Tests, а модули тестирования - в папках /Tests/Complex Tests/ и /Tests/Tree 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
Tree	**1**	/Tests/Tree Tests/ list_unit_1.cpp	Создание дерева и удаление элементов
	2	/Tests/Tree Tests/ list_unit_2.cpp	Удаление из дерева
	3	/Tests/Tree Tests/ list_unit_3.cpp	Вывод дерева в соответствии с обходом
	4	/Tests/Tree Tests/ list_unit_4.cpp	Удаление поддерева и поиск элемента на вхождение
	5	/Tests/Tree Tests/ list_unit_5.cpp	map, where, merge

Приложение

main.cpp:

```
#include "tree.hpp"
#include "complex.hpp"
#include "Tests/tests.hpp"
#include "UI.hpp"

using namespace std;

int main(int argc, const char *
argv[]) {

    run_UI();
    return 0;
}
```

tree.cpp:

```
#include "tree.hpp"

using namespace std;

template <typename T>
Node<T>::Node(){
    data = NULL;
    left = nullptr;
    right = nullptr;
}

template <typename T>
Node<T>::Node(const T x){
    data = x;
    left = nullptr;
    right = nullptr;
}

template <typename T>
Tree<T>::Tree() {
    root = nullptr;
}

//template <typename T>
//Tree<T>::Tree(Node<T> &rt) {
//    root = copy_node(&rt);
//}
//
//template <typename T>
//Tree<T>::Tree(Tree<T> &tr) {
//    root = copy_node(tr.root);
//}

template <typename T>
Tree<T>::~Tree() {
//    delete root;
}

template <typename T>
void Tree<T>::add_node(T x){
    Node<T>* n = new Node<T>(x);
    Node<T>* ptr;
    Node<T>* ptr1 = nullptr;

    n->left = 0;
    n->right = 0;
    n->parent = 0;
    ptr = root;
    while (ptr != 0) {
        ptr1 = ptr;
        if (x < ptr->data)
            ptr = ptr->left;
        else
            ptr = ptr->right;
    }

    n->parent = ptr1;
    if (ptr1 == 0)
        root = n;
    else{
        if(x < ptr1->data )
            ptr1->left=n;
        else
            ptr1->right=n;
    }
}

template <typename T>
void Tree<T>::remove_node(Node<T>* x){
    if (root == nullptr || x ==
nullptr) return;
    if (x == root)
    {
        if (x->left && x->right) {
            Node<T> *max_node =
find_max_node(x->left);
            x->data = max_node->data;
            remove_node(max_node);
            return;
        }
        else if (x->left) {
            root = x->left;
        }
        else if (x->right) {
            root = x->right;
        }
        else {
            root = nullptr;
        }
        free(x);
        clear_node(x);
    }
    else
    {
        if (x->left && x->right){
            Node<T> *max_node =
find_max_node(x->left);
            x->data = max_node->data;
            remove_node(max_node);
            return;
        }
        else if (x->left){
            if (x == x->parent->left)
x->parent->left = x->left;
            else x->parent->right = x-
>left;
        }
        else if (x->right){
            if (x == x->parent->right)
x->parent->right = x->right;
            else x->parent->left = x-
>right;
        }
        else{
            if (x == x->parent->left)
x->parent->left = nullptr;
            else x->parent->right =
nullptr;
        }
        free(x);
        clear_node(x);
    }
}

template <typename T>
void Tree<T>::remove_node(T val){
    Node<T>* n = find_node(val);
    remove_node(n);
}
```

```
template <typename T>
void Tree<T>::clear_node(Node<T> *n){
    n = nullptr;
    return;
}

template <typename T>
string
Tree<T>::post_order_to_str(Node<T>* x)
{
    stringstream ss;
    if (x != 0){
        if (x->left != NULL || x-
>right != NULL) ss<<"[";
        ss<<this->post_order_to_str(x-
>left);
        if (x->left != NULL || x-
>right != NULL) ss<<"][";
        ss<<this->post_order_to_str(x-
>right);
        if (x->left != NULL || x-
>right != NULL) ss<<"]";
        ss<<x->data;
    }
    string s = ss.str();
    return s;
}

template <typename T>
string Tree<T>::post_order_to_str() {
    return post_order_to_str(root);
}

template <typename T>
string
Tree<T>::reverse_post_order_to_str(Nod
e<T>* x) {
    stringstream ss;
    if (x != 0){
        if (x->left != NULL || x-
>right != NULL) ss<<"[";
        ss<<this-
>reverse_post_order_to_str(x->right);
        if (x->left != NULL || x-
>right != NULL) ss<<"][";
        ss<<this-
>reverse_post_order_to_str(x->left);
        if (x->left != NULL || x-
>right != NULL) ss<<"]";
        ss<<x->data;
    }
    string s = ss.str();
    return s;
}

template <typename T>
string
Tree<T>::reverse_post_order_to_str() {
    return
reverse_post_order_to_str(root);
}

template <typename T>
Node<T>* Tree<T>::find_node(T val){
    Node<T>* current_node = root;

    while(current_node != nullptr){
        if (val < current_node->data)
current_node = current_node->left;
        else if (val > current_node-
>data) current_node = current_node-
>right;
        else if (val == current_node-
>data){
            return current_node;
            break;
        }
        if (current_node == nullptr)
return nullptr;
    }
    return nullptr;
}

template <typename T>
bool Tree<T>::is_in_tree(const T val){
    if (find_node(val) == NULL) return
false;
    else return true;
}

template <typename T>
Node<T>*
Tree<T>::find_min_node(Node<T> *n){
    Node<T>* current_node = n;
    while (current_node->left !=
nullptr)
        current_node = current_node-
>left;

    return current_node;
}

template <typename T>
Node<T>*
Tree<T>::find_max_node(Node<T> *n){
    Node<T>* current_node = n;
    while (current_node->right !=
nullptr)
        current_node = current_node-
>right;

    return current_node;
}

template <typename T>
Node<T>* Tree<T>::copy_node(Node<T>
*node_to_copy, Tree<T> &new_tree){
    if (node_to_copy != nullptr)
    {

new_tree.add_node(node_to_copy->data);
        copy_node(node_to_copy->left,
new_tree);
        copy_node(node_to_copy->right,
new_tree);
    }
    return new_tree.root;
}
```

```
template <typename T>
Tree<T> Tree<T>::copy_tree(){
    Tree<T> new_tree;
    Node<T>* n = root;
    copy_node(n,new_tree);

    return new_tree;
}

template <typename T>
Tree<T> Tree<T>::copy_tree(Node<T>
*node_to_copy){
    Tree<T> new_tree;
    copy_node(node_to_copy, new_tree);
    return new_tree;
}

template <typename T>
Node<T>* Tree<T>::get_root(){
    return root;
}

template <typename T>
void Tree<T>::remove_subtree(Node<T>
*n){
    if (n != nullptr)
    {
        remove_subtree(n->left);
        remove_subtree(n->right);
        remove_node(n);
    }
}

template <typename T>
void Tree<T>::remove_subtree(const T
val){
    Node<T>* n;
    n = find_node(val);
    remove_subtree(n);
}

template <typename T>
Node<T>* Tree<T>::where(where_func wf,
Node<T>* n, Tree<T> &new_tree) {
    if (n != nullptr)
    {
        if (wf(n->data))
        {
            new_tree.add_node(n-
>data);
        }
        where(wf,n->left, new_tree);
        where(wf,n->right, new_tree);
    }
    return new_tree.root;
}

template <typename T>
Tree<T> Tree<T>::where(where_func wf){
    Tree<T> new_tree;
    Node<T>* n = root;
    where(wf, n, new_tree);

    return new_tree;
}

template <typename T>
Node<T>* Tree<T>::merge(Node<T> *n,
Tree<T> &new_tree){
    if (n != nullptr)
    {
        new_tree.add_node(n->data);
        merge(n->left, new_tree);
        merge(n->right, new_tree);
    }
    return new_tree.root;
}

template <typename T>
Tree<T> Tree<T>::merge(Tree<T>
&sec_tree){
    Tree<T> new_tree;
    new_tree = copy_tree(root);
    Node<T> *n = sec_tree.root;
    merge(n, new_tree);
    return new_tree;
}

template <typename T>
void Tree<T>::map(Node<T> *n, map_func
mf){
    if (n != nullptr)
    {
        n->data = mf(n->data);
        map(n->left, mf);
        map(n->right, mf);
    }
}
template <typename T>
void Tree<T>::map(map_func mf){
    map(root, mf);
}

template <typename T>
void Tree<T>::print(Node<T>* n, int
level){

    if (n != nullptr){
        print(n->right, level + 1);
        for (int i = 0; i < level; i+
+) cout<<"   ";
        cout<<n->data<<endl;
        print(n->left, level + 1);
    }
}

template <typename T>
void Tree<T>::print(){
    if (root == nullptr) cout<<"\nTree
is empty.\n\n";
    else{
        print(root, get_depth());
        cout<<endl;
    }
}

template <typename T>
int Tree<T>::get_depth(){
    int length = 0;
    return get_depth(root,length);
```

```
}

template <typename T>
int Tree<T>::get_depth(Node<T> *n, int
depth){
    if (n == nullptr)
        return depth;
    return max(get_depth(n->left,
depth + 1), get_depth(n->right, depth
+ 1));
}
```

tree.hpp

```
#ifndef three_hpp
#define three_hpp

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

using namespace std;

template <typename T>
struct Node{
    T data;
    Node<T>* left;
    Node<T>* right;
    Node<T>* parent;

    Node();
    Node(const T);
};

template <typename T>
class Tree{
private:
    Node<T>* root;

    void remove_subtree(Node<T> *);
//вспомогательная функция для удаления
поддерева
    Node<T>* merge(Node<T>*, Tree<T>
&);    //вспомогательная функция для
merge
    void print(Node<T>*, int);
//вспомогательная функция для print
    int get_depth(Node<T>*, int);
//вспомогательная функция для
get_depth

public:
    Tree();
//    Tree(Node<T> &rt);
//    Tree(Tree<T> &tr);
    ~Tree();

    void add_node(T);           //
добаление элемента в дерево
    void remove_node(T);        //
удаление элемента из дерево по
значению
    void remove_node(Node<T>*); //
удаление элемента из дерево по узлу
    void clear_node(Node<T>*);

    string post_order_to_str(Node<T>*
x);          //обход левый-правый-
корень, начиная с x
    string post_order_to_str();
//обход левый-правый-корень, начиная с
корня
    string
reverse_post_order_to_str(Node<T>* x);
//обход правый-левый-корень, начиная с
x
    string
reverse_post_order_to_str();
//обход правый-левый-корень, начиная с
корня

    Node<T>* find_node(T);          //
находит узел по значению
    bool is_in_tree(T);             //
проверка элемента на вхождение
    Node<T>* find_min_node(Node<T>*);
    Node<T>* find_max_node(Node<T>*);

    Node<T>* copy_node(Node<T>*,
Tree<T>&);     //копирует узел
    Tree<T> copy_tree();
//копирует дерево
    Tree<T> copy_tree(Node<T>*);
    Node<T>* get_root();

    void remove_subtree(T
val);       //удаляет поддерево по
значению

    typedef bool (*where_func)(T);
    Node<T>* where(where_func,
Node<T>*, Tree<T>&); //вспомогательная
функция для where
    Tree<T> where(where_func wf);

    Tree<T> merge(Tree<T> &sec_tree);
//прибавляет второе дерево к первому
(слияние)

    typedef T (*map_func)(T);
    void  map(Node<T> *, map_func);
//вспомогательная функция для map
    void map(map_func mf);

    void print();
    int get_depth(); //находит глубину
дерева
};

#include "tree.cpp"

#endif /* three_hpp */
```

UI.cpp

```
#include "UI.hpp"

int cin_int(){
    int N;
```

```
    while (true) {
        cout<<flush;//очистка потока
        if ((cin >> N).good()) return
N;
        if (cin.fail()) {
            cin.clear();//очищает флаг
ошибки, но часть введенной строки при
этом остается
            cout << "Invalid input.
\n";
        }

cin.ignore(numeric_limits<streamsize>:
:max(), '\n');//пропустить все символы
в потоке до \n, после этого можно
начинать считывать новое значение
    }
}

double cin_double(){
    double N;
    while (true) {
        cout<<flush;//очистка потока
        if ((cin >> N).good()) return
N;
        if (cin.fail()) {
            cin.clear();//очищает флаг
ошибки, но часть введенной строки при
этом остается
            cout << "Invalid input.
\n";
        }

cin.ignore(numeric_limits<streamsize>:
:max(), '\n');//пропустить все символы
в потоке до \n, после этого можно
начинать считывать новое значение
    }
}

bool db_check_values(double db){
    if (db > 0) return true;
    else return false;
}

bool com_check_values(Complex db){
    if (db > 0) return true;
    else return false;
}

double db_square(double db){
    return db*db;
}

Complex com_square(Complex com){
    return com*com;
}

void run_UI(){
    Tree<double>::map_func db_mf =
db_square;
    Tree<Complex>::map_func com_mf =
com_square;
    Tree<double> dbTree;
    Tree<Complex> comTree;
    int type = 0;

    int menu_item = 0;
    while (menu_item != 12){
        cout<<"\nPress\n";
        cout<<" 1.  to add item in
tree\n";
        cout<<" 2.  to remove item\n";
        cout<<" 3.  to print tree\n";
        cout<<" 4.  use map function
(square)\n";
        cout<<" 5.  use where function
(remove all non-positive)\n";
        cout<<" 6.  merge\n";
        cout<<" 7.  remove subtree\n";
        cout<<" 8.  search for a
value\n";
        cout<<" 9.  post-order
traversal (LEFT-RIGHT-ROOT) \n";
        cout<<" 10. reverse post-order
traversal (RIGHT-LEFT-ROOT) \n";
        cout<<" 11. to run tests\n";
        cout<<" 12. to exit\n";
        menu_item = cin_int();

        switch (menu_item) {
            case 1:
            {
                cout<<endl<<"Select
type of tree:\n";
                cout<<" 1. Real\n";
                cout<<" 2. Complex\n";
                type = cin_int();

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

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

cout<<endl<<"Select type of tree:\n";
                    cout<<" 1.
Real\n";
                    cout<<" 2.
Complex\n";
                    type = cin_int();
                }

                if (type == DOUBLE) {
                    cout<<"\nEnter
value:\n";

dbTree.add_node(cin_double());
                }
                else {
                    double re = 0;
                    double im = 0;
                    cout<<"\nEnter
real part of value:\n";
                    re = cin_double();
                    cout<<"\nEnter
imaginary part of value:\n";
                    im = cin_double();

comTree.add_node(Complex(re,im));
                }
            }
```

```
                break;

            case 2:
            {
                if (type == DOUBLE) {
                    cout<<"\nEnter
value you want to remove:\n";
                    double val =
cin_double();

dbTree.remove_node(val);
                }
                else if (type ==
COMPLEX){
                    cout<<"\nEnter
value you want to remove:\n";
                    cout<<"\n Enter
real part:\n";
                    double re =
cin_double();
                    cout<<"\n Enter
imaginary part:\n";
                    double im =
cin_double();

comTree.remove_node(Complex(re,im));
                }
                else cout<<"\nTree
doesn't exist\n";
            }
                break;

            case 3:
            {
                if (type == DOUBLE) {
                    cout<<"\nYour
tree:\n";
                    dbTree.print();
                }
                else if (type ==
COMPLEX){
                    cout<<"\nYour
tree:\n";
                    comTree.print();
                }
                else cout<<"\nTree
doesn't exist\n";
            }
                break;

            case 4:
            {
                if (type == DOUBLE) {
                    dbTree.map(db_mf);
                }
                else if (type ==
COMPLEX){


comTree.map(com_mf);
                }
                else cout<<"\nTree
doesn't exist\n";
            }
                break;

            case 5:
            {
                if (type == DOUBLE) {

Tree<double>::where_func wf =
db_check_values;
                    dbTree.where(wf);
                }
                else if (type ==
COMPLEX){

Tree<Complex>::where_func wf =
com_check_values;
                    comTree.where(wf);
                }
                else cout<<"\nTree
doesn't exist\n";
            }
                break;

            case 6:
            {
                if (type == 0)
cout<<"\nTree doesn't exist\n";
                else{
                    int menu_item = 0;
                    Tree<double>
sec_dbTree;
                    Tree<Complex>
sec_comTree;
                    cout<<"\nEnter
second three:\n";
                    cout<<"Do you want
to enter new item?\n";
                    while (menu_item !
= 2) {
                        cout<<"
1.Yes\n";
                        cout<<"
2.No\n";
                        menu_item =
cin_int();
                        while
(menu_item != 1 && menu_item != 2) {

cout<<"Incorrect input.\n";
                            menu_item
= cin_int();
                        }
                        if (menu_item
== 1) {
                            if (type
== DOUBLE) {

cout<<"\nEnter value:\n";

sec_dbTree.add_node(cin_double());
                            }
                            else{
                                double
re = 0;
                                double
im = 0;

cout<<"\nEnter real part of value:\n";
```

```
                                re =
cin_double();

cout<<"\nEnter imaginary part of
value:\n";
                                im =
cin_double();

sec_comTree.add_node(Complex(re,im));
                            }
                        }
                    }

                    if (type ==
DOUBLE) dbTree =
dbTree.merge(sec_dbTree);
                    else comTree =
comTree.merge(sec_comTree);
                }
            }
                break;

            case 7:
            {
                if (type == DOUBLE) {
                    cout<<"\nEnter
value:\n";

dbTree.remove_subtree(cin_double());
                }
                else if (type ==
COMPLEX){
                    cout<<"\nEnter
value:\n";
                    double re = 0;
                    double im = 0;
                    cout<<"\nEnter
real part of value:\n";
                    re = cin_double();
                    cout<<"\nEnter
imaginary part of value:\n";
                    im = cin_double();

comTree.remove_subtree(Complex(re,im))
;
                }
                else cout<<"\nTree
doesn't exist\n";
            }
                break;

            case 8:
            {
                if (type == DOUBLE) {
                    cout<<"\nEnter
value you want to check:\n";
                    if
(dbTree.is_in_tree(cin_double()))
cout<<"\nValue enters the tree\n";
                    else
cout<<"\nValue doesn't enter the
tree\n";
                }
                else if (type ==
COMPLEX){
                    cout<<"\nEnter
value you want to check:\n";
                    double re = 0;
                    double im = 0;
                    cout<<"\nEnter
real part of value:\n";
                    re = cin_double();
                    cout<<"\nEnter
imaginary part of value:\n";
                    im = cin_double();
                    if
(comTree.is_in_tree(Complex(re,im)))
cout<<"\nValue enters the tree\n";
                    else
cout<<"\nValue doesn't enter the
tree\n";
                }
                else cout<<"\nTree
doesn't exist\n";
            }
                break;

            case 9:
            {
                if (type == DOUBLE) {

cout<<dbTree.post_order_to_str();
                }
                else if (type ==
COMPLEX){

cout<<comTree.post_order_to_str();
                }
                else cout<<"\nTree
doesn't exist\n";
            }
                break;

            case 10:
            {
                if (type == DOUBLE) {

cout<<dbTree.reverse_post_order_to_str
();
                }
                else if (type ==
COMPLEX){

cout<<comTree.reverse_post_order_to_st
r();
                }
                else cout<<"\nTree
doesn't exist\n";
            }
                break;

            case 11:
            {
                run_all_tests();
            }
                break;

            case 12:
                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 "tree.hpp"
#include "complex.hpp"
#include "Tests/tests.hpp"

void run_UI();

int cin_int();
double cin_double();

bool check_values(double);
bool check_values(Complex);
double square(double);
Complex square(Complex);

#endif /* UI_hpp */
```

Tests/tests.cpp

```
#include "tests.hpp"

using namespace std;

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_tree_tests();
}


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

void run_tree_tests(){
    run_tree_unit_1();
    run_tree_unit_2();
    run_tree_unit_3();
    run_tree_unit_4();
    run_tree_unit_5();
}
```

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 "Tree Tests/tree_unit_1.hpp"
#include "Tree Tests/tree_unit_2.hpp"
#include "Tree Tests/tree_unit_3.hpp"
#include "Tree Tests/tree_unit_4.hpp"
#include "Tree Tests/tree_unit_5.hpp"


using namespace std;

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

#endif /* tests_hpp */
```

Tests/Tree Tests/tree_unit_1.cpp

```
#include "tree_unit_1.hpp"

using namespace std;

/*
 UNIT 1: Тестирование создания дерева
и добавления элементов

 Список тестов:
 Тест 1: Добавление одного элемента
 Тест 2: Добавление пяти элементов
 Тест 3: Создание пустого дерева
 */

void run_tree_unit_1(){
    cout<<"[TREE 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);

        Tree<double> tr;
        tr.add_node(5);

        if (tr.get_root()->data == 5
&& tr.get_root()->left == nullptr &&
tr.get_root()->left == nullptr)
test_ok = true;
        else test_ok = false;

        test_result(test_ok);
    }

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

        Tree<double> tr;
        tr.add_node(5);
        tr.add_node(2);
        tr.add_node(3);
        tr.add_node(-5);
        tr.add_node(1);

        Node<double>* current_node =
tr.get_root();
        if (current_node->data == 5 &&
current_node->right == nullptr){
            current_node =
current_node->left;
            if (current_node->data ==
2){
                current_node =
current_node->right;
                if (current_node->data
== 3 && current_node->left == nullptr
&& current_node->right == nullptr){
                    current_node =
current_node->parent->left;
                    if (current_node-
>data == -5 && current_node->left ==
nullptr){
                        current_node =
current_node->right;
                        if
(current_node->data == 1 &&
current_node->left == nullptr &&
current_node->right == nullptr)
test_ok = true;
                    }
                }
            }
        }
        else test_ok = false;

        test_result(test_ok);
    }

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

        Tree<double> tr;

        if (tr.get_root() == NULL)
test_ok = true;
        else test_ok = false;

        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/Tree Tests/tree_unit_1.hpp

```
#ifndef tree_unit_1_hpp
#define tree_unit_1_hpp

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

using namespace std;

void run_tree_unit_1(void);

#endif /* tree_unit_1_hpp */
```

Tests/Tree Tests/tree_unit_2.cpp

```
#include "tree_unit_2.hpp"

using namespace std;

/*
 UNIT 2: Тестирование удаления числа
из дерева

 Список тестов:
 Тест 1: Удаление корня из дерева
 Тест 2: Удаление элемента без
потомков
 Тест 3: Удаление элемента с одним
потомком
 Тест 4: Попытка удаления
несуществующего элемента

 */

void run_tree_unit_2(){
    cout<<"[TREE 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);

        Tree<double> tr;
        tr.add_node(5);
        tr.add_node(2);
        tr.add_node(3);
        tr.add_node(-5);
        tr.add_node(1);
        tr.add_node(6);

        tr.remove_node(5);

        Node<double>* current_node =
tr.get_root();
        if (current_node->data == 3){
            current_node =
current_node->right;
            if (current_node->data ==
6 && current_node->left == nullptr &&
current_node->right == nullptr){
                current_node =
current_node->parent->left;
                if (current_node->data
== 2 && current_node->right ==
nullptr){
                    current_node =
current_node->left;
                    if (current_node-
>data == -5 && current_node->left ==
nullptr){
                        current_node =
current_node->right;
                        if
(current_node->data == 1 &&
current_node->left == nullptr &&
current_node->right == nullptr)
test_ok = true;
                    }
                }
            }
        }

        test_result(test_ok);
    }

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

        Tree<double> tr;
        tr.add_node(5);
        tr.add_node(2);
        tr.add_node(3);
        tr.add_node(-5);
        tr.add_node(1);

        tr.remove_node(1);
        Node<double>* current_node =
tr.get_root();
        if (current_node->data == 5 &&
current_node->right == nullptr){
            current_node =
current_node->left;
            if (current_node->data ==
2){
                current_node =
current_node->right;
                if (current_node->data
== 3 && current_node->left == nullptr
&& current_node->right == nullptr){
                    current_node =
current_node->parent->left;
                    if (current_node-
>data == -5 && current_node->left ==
nullptr && current_node->right ==
nullptr) test_ok = true;
                }
            }
        }
        else test_ok = false;

        test_result(test_ok);
    }

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

        Tree<double> tr;
        tr.add_node(5);
        tr.add_node(2);
        tr.add_node(3);
        tr.add_node(-5);
        tr.add_node(1);

        tr.remove_node(-5);
        Node<double>* current_node =
tr.get_root();
        if (current_node->data == 5 &&
current_node->right == nullptr){
            current_node =
current_node->left;
            if (current_node->data ==
2){
                current_node =
current_node->right;
                if (current_node->data
== 3 && current_node->left == nullptr
&& current_node->right == nullptr){
                    current_node =
current_node->parent->left;
                    if (current_node-
>data == 1 && current_node->left ==
nullptr && current_node->right ==
nullptr) test_ok = true;
                }
            }
        }
        else test_ok = false;

        test_result(test_ok);
    }

    //TECT 4
    if (test_ok == true){
```

```
        test_prepare(&test_id,
&test_ok);

        Tree<double> tr;
        tr.add_node(5);
        tr.add_node(2);
        tr.add_node(3);
        tr.add_node(1);

        tr.remove_node(10);

        Node<double>* current_node =
tr.get_root();
        if (current_node->data == 5 &&
current_node->right == nullptr){
            current_node =
current_node->left;
            if (current_node->data ==
2){
                current_node =
current_node->right;
                if (current_node->data
== 3 && current_node->left == nullptr
&& current_node->right == nullptr){
                    current_node =
current_node->parent->left;
                    if (current_node-
>data == 1 && current_node->left ==
nullptr && current_node->right ==
nullptr) test_ok = true;
                }
            }
        }
        else test_ok = false;

        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/Tree Tests/tree_unit_2.hpp

```
#ifndef tree_unit_2_hpp
#define tree_unit_2_hpp

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

using namespace std;

void run_tree_unit_2(void);

#endif /* tree_unit_2_hpp */
```

Tests/Tree Tests/tree_unit_3.cpp

```
#include "tree_unit_3.hpp"

using namespace std;

/*
 UNIT 3: Тестирование вывода дерева в
строку в соответствии с обходом

 Список тестов:
 Тест 1: Вывод по обходу ЛПК
 Тест 2: Вывод по обходу ПЛК
 */

void run_tree_unit_3(){
    cout<<"[TREE 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);

        Tree<double> tr;
        tr.add_node(5);
        tr.add_node(2);
        tr.add_node(3);
        tr.add_node(-5);
        tr.add_node(1);
        tr.add_node(6);

        string s =
tr.post_order_to_str();
        if (s == "[[[][1]-5][3]2]
[6]5") test_ok = true;

        test_result(test_ok);
    }

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

        Tree<double> tr;
        tr.add_node(5);
        tr.add_node(2);
        tr.add_node(3);
        tr.add_node(-5);
        tr.add_node(1);
        tr.add_node(6);

        string s =
tr.reverse_post_order_to_str();
        if (s == "[6][[3][[1]
[]-5]2]5") test_ok = true;

        test_result(test_ok);
    }
```

```
    if (test_id == 2 && 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/Tree Tests/tree_unit_3.hpp

```
#ifndef tree_unit_3_hpp
#define tree_unit_3_hpp

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

using namespace std;

void run_tree_unit_3(void);

#endif /* tree_unit_3_hpp */
```

Tests/Tree Tests/tree_unit_4.cpp

```
#include "tree_unit_4.hpp"

using namespace std;

/*
 UNIT 4: Тестирование удаления
поддерева и поиска на вхождение
элемента

 Список тестов:
 Тест 1: Извлечение поддерева из
дерева
 Тест 2: Поиск на вхождение элемента
 */

void run_tree_unit_4(){
    cout<<"[TREE 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);

        Tree<double> tr;
        tr.add_node(5);
        tr.add_node(2);
        tr.add_node(3);
        tr.add_node(-5);
        tr.add_node(1);
        tr.add_node(6);

        tr.remove_subtree(-5);

        Node<double>* current_node =
tr.get_root();
        if (current_node->data == 5){
            current_node =
current_node->right;
            if (current_node->data ==
6 && current_node->left == nullptr &&
current_node->right == nullptr){
                current_node =
current_node->parent->left;
                if (current_node->data
== 2 && current_node->left == nullptr)
{
                    current_node =
current_node->right;
                    if (current_node-
>data == 3 && current_node->left ==
nullptr && current_node->right ==
nullptr) test_ok = true;
                }
            }
        }

        test_result(test_ok);
    }

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

        Tree<double> tr;
        tr.add_node(5);
        tr.add_node(2);
        tr.add_node(3);
        tr.add_node(-5);
        tr.add_node(1);
        tr.add_node(6);

        if (tr.is_in_tree(3) && !
tr.is_in_tree(10)) test_ok = true;

        test_result(test_ok);
    }

    if (test_id == 2 && 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/Tree Tests/tree_unit_4.hpp

```
#ifndef tree_unit_4_hpp
#define tree_unit_4_hpp

#include <stdio.h>
#include "tree.hpp"
```

```
#include "tests.hpp"

using namespace std;

void run_tree_unit_4(void);

#endif /* tree_unit_4_hpp */
```

Tests/Tree Tests/tree_unit_5.cpp

```
#include "tree_unit_5.hpp"

using namespace std;

/*
 UNIT 5: Тестирование map, where и
merge

 Список тестов:
 Тест 1: map
 Тест 2: where (результат функции
равен истине только когда число
положительно)
 Тест 3: merge
 */

bool check_values(double db){
    if (db > 0) return true;
    else return false;
}

double square(double db){
    return db*db;
}

void run_tree_unit_5(){
    cout<<"[TREE 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);

        Tree<double>::map_func mf =
square;
        Tree<double> tr;
        tr.add_node(5);
        tr.add_node(2);
        tr.add_node(3);
        tr.add_node(-5);
        tr.add_node(1);
        tr.add_node(6);

        tr.map(mf);

        Node<double>* current_node =
tr.get_root();
        if (current_node->data == 25){
            current_node =
current_node->right;
            if (current_node->data ==
36 && current_node->left == nullptr &&
current_node->right == nullptr){
                current_node =
current_node->parent->left;
                if (current_node->data
== 4){
                    current_node =
current_node->right;
                    if (current_node-
>data == 9 && current_node->left ==
nullptr && current_node->right ==
nullptr) {
                        current_node =
current_node->parent->left;
                        if
(current_node->data == 25 &&
current_node->left == nullptr){

current_node = current_node->right;
                            if
(current_node->data == 1 &&
current_node->left == nullptr &&
current_node->right == nullptr)
test_ok = true;
                        }
                    }
                }
            }
        }

        test_result(test_ok);
    }

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

        Tree<double>::where_func wf =
check_values;
        Tree<double> tr;
        tr.add_node(5);
        tr.add_node(2);
        tr.add_node(3);
        tr.add_node(-5);
        tr.add_node(-1);
        tr.add_node(6);

        Tree<double> new_tree =
tr.where(wf);

        Node<double>* current_node =
new_tree.get_root();
        if (current_node->data == 5){
            current_node =
current_node->right;
            if (current_node->data ==
6 && current_node->left == nullptr &&
current_node->right == nullptr){
                current_node =
current_node->parent->left;
                if (current_node->data
== 2 && current_node->left == nullptr)
{
```

```
                    current_node =
current_node->right;
                    if (current_node-
>data == 3 && current_node->left ==
nullptr && current_node->right ==
nullptr) test_ok = true;
                }
            }
        }

        test_result(test_ok);
    }

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

        Tree<double> first_tr;
        first_tr.add_node(5);
        first_tr.add_node(2);
        first_tr.add_node(3);
        first_tr.add_node(1);

        Tree<double> second_tr;
        second_tr.add_node(7);
        second_tr.add_node(0);
        second_tr.add_node(-2);
        second_tr.add_node(4);

        Tree<double> tr =
first_tr.merge(second_tr);

        Node<double>* current_node =
tr.get_root();
        if (current_node->data == 5){
            current_node =
current_node->right;
            if (current_node->data ==
7 && current_node->left == nullptr &&
current_node->right == nullptr){
                current_node =
current_node->parent->left;
                if (current_node->data
== 2){
                    current_node =
current_node->right;
                    if (current_node-
>data == 3 && current_node->left ==
nullptr){
                        current_node =
current_node->right;
                        if
(current_node->data == 4 &&
current_node->left == nullptr &&
current_node->right == nullptr){

current_node = current_node->parent-
>parent->left;
                            if
(current_node->data == 1 &&
current_node->right == nullptr){

current_node = current_node->left;
                                if
(current_node->data == 0 &&
current_node->right == nullptr){

current_node = current_node->left;
                                    if
(current_node->data == -2 &&
current_node->left == nullptr &&
current_node->right == nullptr)
test_ok = true;
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }

        test_result(test_ok);
    }

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

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

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

Tests/Tree Tests/tree_unit_5.hpp

```
#ifndef tree_unit_5_hpp
#define tree_unit_5_hpp

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

using namespace std;

void run_tree_unit_5(void);

bool check_values(double);
double square(double);

#endif /* tree_unit_5_hpp */
```

Tests/Complex Tests/complex_unit_1.cpp

```
#include "complex_unit_1.hpp"

using namespace std;

/*
```

```
 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 3: 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);
    }

    //TECT 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);
    }

    //TECT 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"

using namespace std;

void run_complex_unit_1(void);

#endif /* complex_unit_1_hpp */
```

Tests/Complex Tests/complex_unit_2.cpp

```
#include "complex_unit_2.hpp"

using namespace std;

/*
 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;

    //TECT 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);
    }

    //TECT 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);
    }

    //TECT 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);
    }

    //TECT 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"

using namespace std;

void run_complex_unit_2(void);

#endif /* complex_unit_2_hpp */
```

Tests/Complex Tests/complex_unit_3.cpp

```
#include "complex_unit_3.hpp"

using namespace std;

/*
 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;

    //TECT 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);
    }

    //TECT 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"

using namespace std;

void run_complex_unit_3(void);

#endif /* complex_unit_3_hpp */
```