**Московский авиационный институт**

**(Национальный исследовательский университет)**

Институт: «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

Дисциплина: «Объектно-ориентированное программирование»

**Лабораторная работа № 5**

Тема: Основы работы с коллекциями: итераторы

Студент: Кочисов Алексей

Группа: М8О-201Б-20

Преподаватель: Чернышов Л.Н.

Дата:

Оценка:

Москва, 2020

1. **Постановка задачи**

Разработать программу на языке C++ согласно варианту задания. Программа на C++ должна собираться с помощью системы сборки CMake. Программа должна получать данные из стандартного ввода и выводить данные в стандартный вывод.

Создать шаблон динамической коллекцию, согласно варианту задания:

1. Коллекция должна быть реализована с помощью умных указателей (std::shared\_ptr, std::weak\_ptr). Опционально использование std::unique\_ptr;
2. В качестве параметра шаблона коллекция должна принимать тип данных - фигуры;
3. Реализовать forward\_iterator по коллекции;
4. Коллекция должны возвращать итераторы begin() и end();
5. Коллекция должна содержать метод вставки на позицию итератора insert(iterator);
6. Коллекция должна содержать метод удаления из позиции итератора erase(iterator);
7. При выполнении недопустимых операций (например выход за границы коллекции или удаление несуществующего элемента) необходимо генерировать исключения;
8. Итератор должен быть совместим со стандартными алгоритмами (например, std::count\_if)
9. Коллекция должна содержать метод доступа:

* Стек – pop, push, top;
* Очередь – pop, push, top;
* Список, Динамический массив – доступ к элементу по оператору [];

1. Реализовать программу, которая:

* Позволяет вводить с клавиатуры фигуры (с типом int в качестве параметра шаблона фигуры) и добавлять в коллекцию;
* Позволяет удалять элемент из коллекции по номеру элемента;
* Выводит на экран введенные фигуры с помощью std::for\_each;
* Выводит на экран количество объектов, у которых площадь меньше заданной (с помощью std::count\_if);

**Вариант №13:**

|  |  |
| --- | --- |
| **Фигура** | **Контейнер** |
| Ромб | Список |

**2. Описание программы**

Программа предназначена для работы с одним типом равносторонних фигур – ромбом. Ромб задается через координату центра ромба (точка пересечения диагоналей), и две диагонали. Программа выводит координаты центра и всех вершин, а также может вычислить площадь фигуры. Данная программа в первую очередь предназначена для демонстрации возможностей моей реализации коллекции (списка), поэтому все фигуры, которые вводит пользователь добавляются в список. Для показа всех остальных вариантов взаимодействия со списком был реализован пользовательский интерфейс. Итак пользователь может:

1. Добавить ромб (далее выводиться уточняющее подменю);

- добавить в начало списка,

- добавить в конец списка,

- добавить в список по индексу (далее пользователя просят ввести индекс).

2. Удалить ромб(далее выводиться уточняющее подменю);

- удалить из начала списка,

- удалить из конца списка,

- удалить из списка по индексу (далее пользователя просят ввести индекс).

3. Вывести данные о введенном ромбе (далее выводиться уточняющее подменю);

- о первом ромбе в списке,

- о последнем ромбе в списке,

- об определенном ромбе по индексу (далее пользователя просят ввести индекс).

4. Вывести данные обо всех ромбах в списке;

5. Вывести кол-во ромбов, которые имеют площадь меньше, чем… (далее пользователя просят ввести число, оно не должно быть отрицательным, иначе программа завершит свою работу);

6. Завершить работу программы.

Пользователь вводит цифру соответствующую пункту меню. При добавлении фигуры нужно ввести указанные выше начальные данные.

**Тестовые примеры**

**Test\_01.txt**

2 1 2 2 2 3 4

3 1 2 3 2 3 -1

4

5 -1

**Test\_02.txt**

1 1 3 5 100 1

1 2 0 0 200 2

1 3 2 1 1 40 2

3 1

3 2

3 3 1

4

5 100

2 3 1

2 2

2 1

4

6

**4. Листинг программы**

**Main.ccp**

#include <iostream>

#include <vector>

#include <string>

#include <iomanip>

#include <cmath>

#include <algorithm>

#include "list.hpp"

#include "rhombus.hpp"

void AddSubOptions() {

std::cout << "Add suboptions: " << std::endl;

std::cout << "1. Add rhombus at the beginning of the list" << std::endl;

std::cout << "2. Add rhombus at the end of the list" << std::endl;

std::cout << "3. Add rhombus to the list by index" << std::endl;

std::cout << std::endl;

}

void DeleteSubOptions() {

std::cout << "Delete suboptions: " << std::endl;

std::cout << "1. Delete rhombus from the beginning of the list" << std::endl;

std::cout << "2. Delete rhombus from the end of the list" << std::endl;

std::cout << "3. Delete rhombus from the list by index" << std::endl;

std::cout << std::endl;

}

void PrintSubOptions() {

std::cout << "Print suboptions: " << std::endl;

std::cout << "1. Print the first rhombus in the list" << std::endl;

std::cout << "2. Print the last rhombus in the list" << std::endl;

std::cout << "3. Print the rhombus from the list by index" << std::endl;

std::cout << std::endl;

}

void Options() {

std::cout << "Options: " << std::endl;

std::cout << "1. Add rhombus" << std::endl;

std::cout << "2. Delete rhombus" << std::endl;

std::cout << "3. Print one rhombus in the list" << std::endl;

std::cout << "4. Print all rhombuses in the list" << std::endl;

std::cout << "5. Count rhombuses with area less than..." << std::endl;

std::cout << "6. Exit" << std::endl;

std::cout << std::endl;

}

int main() {

Options();

List<Rhombus<int>> MyList;

Rhombus<int> r;

int option, suboption, index;

double area;

std::cout << "Select option: ";

while (std::cin >> option) {

if (option == 1) { // Add rhombus

AddSubOptions();

std::cout << "Select add suboption: ";

std::cin >> suboption;

switch (suboption) {

case 1: // Add rhombus at the beginning of the list

std::cin >> r.center.first >> r.center.second >> r.diag1 >> r.diag2;

MyList.PushFront(r);

std::cout << "Rhombus succesfully added" << std::endl;

break;

case 2: // Add rhombus at the end of the list

std::cin >> r.center.first >> r.center.second >> r.diag1 >> r.diag2;

MyList.PushBack(r);

std::cout << "Rhombus succesfully added" << std::endl;

break;

case 3: // Add rhombus to the list by index

std::cin >> index;

if (MyList.Length() + 1 < index || index < 0) {

std::cout << "Index is out of range!" << std::endl;

break;

}

std::cin >> r.center.first >> r.center.second >> r.diag1 >> r.diag2;

MyList.InsertByNumber(index, r);

std::cout << "Rhombus succesfully added" << std::endl;

break;

}

}

else if (option == 2) { // Delete rhombus

DeleteSubOptions();

std::cout << "Select remove suboption: ";

std::cin >> suboption;

switch (suboption) {

case 1: // Delete rhombus from the beginning of the list

if (MyList.Length() == 0) {

std::cout << "List is empty!" << std::endl;

break;

}

MyList.PopFront();

std::cout << "Rhombus succesfully removed" << std::endl;

break;

case 2: // Delete rhombus from the end of the list

if (MyList.Length() == 0) {

std::cout << "List is empty!" << std::endl;

break;

}

MyList.PopBack();

std::cout << "Rhombus succesfully removed" << std::endl;

break;

case 3: // Delete rhombus from the list by index

std::cin >> index;

if (MyList.Length() + 1 < index || index < 0) {

std::cout << "Index is out of range!" << std::endl;

break;

}

if (MyList.Length() == 0) {

std::cout << "List is empty!" << std::endl;

break;

}

MyList.EraseByNumber(index);

std::cout << "Rhombus succesfully removed" << std::endl;

break;

}

}

else if (option == 3) { // Print one rhombus in the list

PrintSubOptions();

std::cout << "Select print suboption: ";

std::cin >> suboption;

switch (suboption) {

case 1: // Print the first rhombus in the list

if (MyList.Length() == 0) {

std::cout << "List is empty!" << std::endl;

break;

}

Print(MyList.Front());

break;

case 2: // Print the last rhombus in the list

if (MyList.Length() == 0) {

std::cout << "List is empty!" << std::endl;

break;

}

Print(MyList.Back());

break;

case 3: // Print the rhombus from the list by index

std::cin >> index;

if (MyList.Length() + 1 < index || index < 0) {

std::cout << "Index is out of range!" << std::endl;

break;

}

if (MyList.Length() == 0) {

std::cout << "List is empty!" << std::endl;

break;

}

Print(MyList[index]);

break;

}

}

else if (option == 4) { // Print all rhombuses in the list

if (MyList.Length() == 0)

std::cout << "List is empty!" << std::endl;

std::for\_each(MyList.begin(), MyList.end(), [](const Rhombus<int> &r) {

Print(r);

});

}

else if (option == 5) { // Count rhombuses with area less than...

std::cout << "Please, enter the area that you want to compare: ";

std::cin >> area;

if (area < 0) {

std::cout << "Area can't be negative!" << std::endl;

break;

}

std::cout << std::count\_if(MyList.begin(), MyList.end(), [area](const Rhombus<int> &r) {

return area > CalcArea(r);

}) << std::endl;

}

else if (option == 6) { // Exit

break;

}

else // Wrong option

std::cout << "There is no such option, please try again!" << std::endl;

std::cout << std::endl;

std::cout << "Select option: "; // Repeat input

}

return 0;

}

**List.hpp**

#ifndef LIST\_HPP

#define LIST\_HPP

#include <iterator>

#include <memory>

template <class T>

class List {

private:

struct Element; // forward-declaration

size\_t size = 0; // size of the list

public:

List() = default;

class forward\_iterator {

public:

using value\_type = T;

using reference = value\_type &;

using pointer = value\_type \*;

using difference\_type = std::ptrdiff\_t;

using iterator\_category = std::forward\_iterator\_tag;

forward\_iterator(Element \*ptr);

T &operator\*();

forward\_iterator &operator++();

forward\_iterator operator++(int);

bool operator==(const forward\_iterator &other) const;

bool operator!=(const forward\_iterator &other) const;

private:

Element \*it\_ptr;

friend List;

};

forward\_iterator begin();

forward\_iterator end();

void PushBack(const T &value); // Add element at the beginning of the list

void PushFront(const T &value); // Add element to the end of the list

T &Front(); //Get element from the beginning of the list

T &Back(); //Get the element from the end of the list

void PopBack(); //Remove element from the end of the list

void PopFront(); //Remove element from the beginning of the list

size\_t Length(); //Get size of the list

bool Empty(); //Check emptiness of the list

void EraseByIterator(forward\_iterator d\_it); //Remove element by iterator

void EraseByNumber(size\_t N); //Remove element by number

void InsertByIterator(forward\_iterator ins\_it, T &value); //Add element by iterator

void InsertByNumber(size\_t N, T &value); //Add element by number

List &operator=(const List &other);

T &operator[](size\_t index);

private:

struct Element {

T value;

std::unique\_ptr<Element> next\_element;

Element \*prev\_element = nullptr;

Element(const T &value\_) : value(value\_) {}

forward\_iterator Next();

};

std::unique\_ptr<Element> head;

Element \*tail = nullptr;

};

template <class T>

typename List<T>::forward\_iterator List<T>::begin() {

return forward\_iterator(head.get());

}

template <class T>

typename List<T>::forward\_iterator List<T>::end() {

return forward\_iterator(nullptr);

}

template <class T>

size\_t List<T>::Length() {

return size;

}

template <class T>

bool List<T>::Empty() {

return Length() == 0;

}

template <class T>

void List<T>::PushBack(const T &value) {

if (!size) {

head = std::make\_unique<Element>(value);

tail = head.get();

size++;

return;

}

tail->next\_element = std::make\_unique<Element>(value);

Element \*temp = tail;

tail = tail->next\_element.get();

tail->prev\_element = temp;

size++;

}

template <class T>

void List<T>::PushFront(const T &value) {

size++;

std::unique\_ptr<Element> tmp = std::move(head);

head = std::make\_unique<Element>(value);

head->next\_element = std::move(tmp);

if (head->next\_element != nullptr)

head->next\_element->prev\_element = head.get();

if (size == 1) {

tail = head.get();

}

if (size == 2) {

tail = head->next\_element.get();

}

}

template <class T>

void List<T>::PopFront() {

if (size == 1) {

head = nullptr;

tail = nullptr;

size--;

return;

}

head = std::move(head->next\_element);

head->prev\_element = nullptr;

size--;

}

template <class T>

void List<T>::PopBack() {

if (tail->prev\_element) {

Element \*tmp = tail->prev\_element;

tail->prev\_element->next\_element = nullptr;

tail = tmp;

}

else {

head = nullptr;

tail = nullptr;

}

size--;

}

template <class T>

T &List<T>::Front() {

if (size == 0) {

throw std::logic\_error("error: list is empty");

}

return head->value;

}

template <class T>

T &List<T>::Back() {

if (size == 0) {

throw std::logic\_error("error: list is empty");

}

forward\_iterator i = this->begin();

while (i.it\_ptr->Next() != this->end()) {

i++;

}

return \*i;

}

template <class T>

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

if (this == &other)

return \*this;

size = other.size;

head = std::move(other.head);

return \*this;

}

template <class T>

void List<T>::EraseByIterator(List<T>::forward\_iterator d\_it) {

forward\_iterator i = this->begin(), end = this->end();

if (d\_it == end)

throw std::logic\_error("error: out of range");

if (d\_it == this->begin()) {

this->PopFront();

return;

}

if (d\_it.it\_ptr == tail) {

this->PopBack();

return;

}

if (d\_it.it\_ptr == nullptr)

throw std::logic\_error("error: out of range");

auto temp = d\_it.it\_ptr->prev\_element;

std::unique\_ptr<Element> temp1 = std::move(d\_it.it\_ptr->next\_element);

d\_it.it\_ptr = d\_it.it\_ptr->prev\_element;

d\_it.it\_ptr->next\_element = std::move(temp1);

d\_it.it\_ptr->next\_element->prev\_element = temp;

size--;

}

template <class T>

void List<T>::EraseByNumber(size\_t N) {

forward\_iterator it = this->begin();

for (size\_t i = 0; i < N; ++i) {

++it;

}

this->EraseByIterator(it);

}

template <class T>

void List<T>::InsertByIterator(List<T>::forward\_iterator ins\_it, T &value) {

std::unique\_ptr<Element> tmp = std::make\_unique<Element>(value);

forward\_iterator i = this->begin();

if (ins\_it == this->begin()) {

this->PushFront(value);

return;

}

if (ins\_it.it\_ptr == nullptr) {

this->PushBack(value);

return;

}

tmp->prev\_element = ins\_it.it\_ptr->prev\_element;

ins\_it.it\_ptr->prev\_element = tmp.get();

tmp->next\_element = std::move(tmp->prev\_element->next\_element);

tmp->prev\_element->next\_element = std::move(tmp);

size++;

}

template <class T>

void List<T>::InsertByNumber(size\_t N, T &value) {

forward\_iterator it = this->begin();

for (size\_t i = 0; i < N; ++i) {

++it;

}

this->InsertByIterator(it, value);

}

template <class T>

typename List<T>::forward\_iterator List<T>::Element::Next() {

return forward\_iterator(this->next\_element.get());

}

template <class T>

List<T>::forward\_iterator::forward\_iterator(List<T>::Element \*ptr) {

it\_ptr = ptr;

}

template <class T>

T &List<T>::forward\_iterator::operator\*() {

return this->it\_ptr->value;

}

template <class T>

T &List<T>::operator[](size\_t index) {

if (index < 0 || index >= size) {

throw std::logic\_error("error: out of range");

}

forward\_iterator it = this->begin();

for (size\_t i = 0; i < index; i++) {

it++;

}

return \*it;

}

template <class T>

typename List<T>::forward\_iterator &List<T>::forward\_iterator::operator++() {

if (it\_ptr == nullptr)

throw std::logic\_error("error: out of range");

\*this = it\_ptr->Next();

return \*this;

}

template <class T>

typename List<T>::forward\_iterator List<T>::forward\_iterator::operator++(int) {

forward\_iterator old = \*this;

++\*this;

return old;

}

template <class T>

bool List<T>::forward\_iterator::operator==(const forward\_iterator &other) const

{

return it\_ptr == other.it\_ptr;

}

template <class T>

bool List<T>::forward\_iterator::operator!=(const forward\_iterator &other) const {

return it\_ptr != other.it\_ptr;

}

#endif /\* LIST\_HPP \*/}

**Rhombus.hpp**

#ifndef RHOMBUS\_HPP

#define RHOMBUS\_HPP

#include <tuple>

template<class T>

struct Rhombus {

std::pair<T, T> center;

double diag1;

double diag2;

};

// Calculates area of rhombus

template<class T>

double CalcArea(T &r) {

return (r.diag1 + r.diag2) \* 0.5;

}

// Prints rhombus

template<class T>

void Print(T &r) {

std::cout.precision(2);

std::cout << "Rhombus:" << std::endl;

std::cout << "1. Center: (" << r.center.first << ", " << r.center.second << ")" << std::endl;

std::cout << "2. Coordinates: (";

std::cout << r.center.first + r.diag1 \* 0.5 << "; " << r.center.second << "), (";

std::cout << r.center.first << "; " << r.center.second + r.diag2 \* 0.5 << "), (";

std::cout << r.center.first - r.diag1 \* 0.5 << "; " << r.center.second << "), (";

std::cout << r.center.first << "; " << r.center.second - r.diag2 \* 0.5 << ")" << std::endl;

std::cout << "3. Area of figure: " << CalcArea(r) << std::endl;

}

#endif /\* RHOMBUS\_HPP \*/

**CMakeLists.txt**

cmake\_minimum\_required(VERSION 3.17)

project(oop\_lab5)

set(CMAKE\_CXX\_STANDARD 17)

add\_executable(oop\_lab5 main.cpp)

**5. Выводы**

В ходе лабораторной работы я изучил основы работы с коллекциями в С++, ознакомился с шаблоном проектирования «Итератора».