Московский авиационный институт
(Национальный исследовательский университет)

Институт: «Информационные технологии и прикладная математика»
Кафедра: 806 «Вычислительная математика и программирование»
Дисциплина: «Объектно-ориентированное программирование»

Лабораторная работа № 6

Тема: Основы работы с коллекциями: аллокаторы

Студент: Мариничев Иван Александрович
Группа: 80-208
Преподаватель: Чернышов Л.Н.
Дата: 7.12.20
Оценка:

Москва, 2020

1. Постановка задачи. Вариант 13

Разработать шаблон класса **ромб**. Параметром шаблона должен являться скалярный тип данных задающий тип данных для оси координат. Класс должен иметь публичные поля. Фигура являются фигурой вращения, т.е. равносторонней. Для хранения координат фигур необходимо использовать шаблон **std::pair**.

Например:

```
template <class T>
struct Square{
    using vertex_t = std::pair<T,T>;
    vertex_t a,b,c,d;
};
```

Создать шаблон динамической коллекции, а именно списка:

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;
3. Коллекция должна содержать метод доступа:
   – доступ к элементу по оператору **[]**;
4. Реализовать аллокатор (**Динамический массив**), который выделяет фиксированный размер памяти (количество блоков памяти – является параметром шаблона аллокатора). Внутри аллокатор должен хранить указатель на используемый блок памяти и динамическую коллекцию указателей на свободные блоки. Динамическая коллекция должна соответствовать варианту задания (**Список**);
5. Коллекция должна использовать аллокатор для выделения и освобождения памяти для своих элементов.
6. Аллокатор должен быть совместим с контейнерами **std::map** и **std::list** (опционально – **vector**).
7. Реализовать программу, которая:
   - Позволяет вводить с клавиатуры фигуры (с типом int в качестве параметра шаблона фигуры) и добавлять в коллекцию использующую аллокатор;
   - Позволяет удалять элемент из коллекции по номеру элемента;
   - Выводит на экран введенные фигуры с помощью **std::for_each**;

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, параллельные осям Ox и Oy. 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. Завершить работу программы.

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

3. Руководство по использованию программы.

Таблица 1 – Функции и структуры файла list.hpp

Название	Аргументы	Описание

class List	private:
	struct Element;
	size_t size = 0;
	public:
	List() = default;
	class forward_iterator
	forward_iterator begin();
	forward_iterator end();
	void PushBack(const T &value);
	void PushFront(const T &value);
	T &Front(); T &Back();
	void PopBack();
	void PopFront
	size_t Length();
	bool Empty();
	void EraseByIterator(forward_iterator d_it);
	void EraseByNumber(size_t N);
	void InsertByIterator(forward_iterator ins_it, T &value);
	void InsertByNumber(size_t N, T &value);
	List &operator=(const List &other);
	T &operator[](size_t index);
	private:
	struct Element
	std::unique_ptr<Element> head;
	Element *tail = nullptr;
		Класс список

class forward_iterator	public:
	using value_type = T;
	using reference = value_type &;
	using pointer = value_type *;
	using difference_type = std::ptrdiff_t;
		Класс forward_iterator

	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;	

struct Element	T value; std::unique_ptr<Element> next_element; Element *prev_element = nullptr; Element(const T &value_) : value(value_) {} forward_iterator Next();	Структура элемент списка
template <class T> typename List<T>::forward_iterator List<T>::begin()		Указатель на голову списка
template <class T> typename List<T>::forward_iterator List<T>::end()		Указатель на хвост списка

template <class T> size_t List<T>::Length()		Метод для нахождения длины списка
template <class T> bool List<T>::Empty()		Метод для для проверки на пустоту
template <class T> void List<T>::PushBack ()	const T &value	Метод добавления элемента в конец списка
template <class T> void List<T>::PushFront ()	const T &value	Метод добавления элемента в начало списка
template <class T> void List<T>::PopFront( )		Удалить элемент из начала списка

template <class T> void List<T>::PopBack( )		Удалить элемент из конца списка
template <class T> T &List<T>::Front()		Вывести элемент из начала списка
template <class T> T &List<T>::Back()		Вывести элемент из конца списка
template <class T> List<T> &List<T>::operator =()	const List<T> &other	Перегрузка оператора копирования
template <class T> void List<T>::EraseByIterator()	List<T>::forward_iterator d_it	Удалить элемент из списка по итератору

template <class T> void List<T>::EraseByNumber()	size_t N	Удалить элемент из списка по числу
template <class T> void List<T>::InsertByIterator()	List<T>::forward_iterator ins_it, T &value	Вставить элемент в список по итератору
template <class T> void List<T>::InsertByNumber()	size_t N, T &value	Вставить элемент в список по числу
template <class T> typename List<T>::forward_iterator List<T>::Element::Next()		Указатель на следующий элемент в списке
template <class T> List<T>::forward_iterator::forward_iterator()	List<T>::Element *ptr	Конструктор forward_iterator

template <class T> T &List<T>::forward _iterator::operator*( )		Перегрузка оператора *
template <class T> T &List<T>::operator []()	size_t index	Перегрузка оператора []
template <class T> typename List<T>::forward_it erator &List<T>::forward _iterator::operator+ +()		Перегрузка оператора ++ для указателей
template <class T> typename List<T>::forward_it erator List<T>::forward_it erator::operator++()	int	Перегрузка оператора ++ для чисел
template <class T> bool List<T>::forward_it erator::operator==()	const forward_iterator &other	Перегрузка оператора ==

template <class T> bool List<T>::forward_iterator::operator!=()	const forward_iterator &other	Перегрузка оператора !=

Таблица 2 – Функции и структуры файла rhombus.hpp

Название	Аргументы	Описание
struct Rhombus	std::pair<T, T> center; double diag1; double diag2;	Структура ромб
template<class T> double CalcArea()	T &r	Функция для вычисления площади ромба
template<class T> void Print()	T &r	Функция выводящая всю информацию о ромбе

Таблица 3 – Функции файла main.cpp

Название	Аргументы	Описание

void AddSubOptions()		Функция вывода подменю для добавления ромба
void DeleteSubOptions()		Функция вывода подменю для удаления ромба
void PrintSubOptions()		Функция вывода подменю для демонстрации информации о ромбе
void Options()		Функция, выводящая основное меню
int main()		Основная функция, в которой происходит демонстрация всех возможностей программы

Результаты выполнения тестов

Таблица 4 – Тесты и результаты работы с ними

Название	Входные	Результат

тестового файла	данные	
test_01.txt	2 1 2 2 2 3 4 3 1 2 3 2 3 -1 4 5	Options: 1. Add rhombus 2. Delete rhombus 3. Print one rhombus in the list 4. Print all rhombuses in the list 5. Exit Select option: Delete suboptions: 1. Delete rhombus from the beginning of the list 2. Delete rhombus from the end of the list 3. Delete rhombus from the list by index Select remove suboption: List is empty! Select option: Delete suboptions: 1. Delete rhombus from the beginning of the list 2. Delete rhombus from the end of the list 3. Delete rhombus from the list by index Select remove suboption: List is empty! Select option: Delete suboptions: 1. Delete rhombus from the beginning of the list 2. Delete rhombus from the end of the list 3. Delete rhombus from the list by index Select remove suboption: Index is out of range! Select option: Print suboptions: 1. Print the first rhombus in the list 2. Print the last rhombus in the list 3. Print the rhombus from the list by index Select print suboption: List is empty! Select option: Delete suboptions: 1. Delete rhombus from the beginning of the list 2. Delete rhombus from the end of the list 3. Delete rhombus from the list by index Select remove suboption: Index is out of range! Select option: Print suboptions: 1. Print the first rhombus in the list 2. Print the last rhombus in the list 3. Print the rhombus from the list by index Select print suboption: Select option: List is empty!

		Select option:
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 2 3 1 2 2 2 1 4 5	Options: 1. Add rhombus 2. Delete rhombus 3. Print one rhombus in the list 4. Print all rhombuses in the list 5. Exit Select option: Add suboptions: 1. Add rhombus at the beginning of the list 2. Add rhombus at the end of the list 3. Add rhombus to the list by index Select add suboption: Rhombus successfully added Select option: Add suboptions: 1. Add rhombus at the beginning of the list 2. Add rhombus at the end of the list 3. Add rhombus to the list by index Select add suboption: Rhombus successfully added Select option: Add suboptions: 1. Add rhombus at the beginning of the list 2. Add rhombus at the end of the list 3. Add rhombus to the list by index Select add suboption: Rhombus successfully added Select option: Print suboptions: 1. Print the first rhombus in the list 2. Print the last rhombus in the list 3. Print the rhombus from the list by index Select print suboption: Rhombus: 1. Center: (3, 5) 2. Coordinates: (53; 5), (3; 5.5), (-47; 5), (3; 4.5) 3. Area of figure: 50 Select option: Print suboptions: 1. Print the first rhombus in the list 2. Print the last rhombus in the list 3. Print the rhombus from the list by index Select print suboption: Rhombus: 1. Center: (1, 1) 2. Coordinates: (21; 1), (1; 2), (-19; 1), (1; 0) 3. Area of figure: 21 Select option: Print suboptions: 1. Print the first rhombus in the list 2. Print the last rhombus in the list 3. Print the rhombus from the list by index

		Select print suboption: Rhombus: 1. Center: (0, 0) 2. Coordinates: (1e+02; 0), (0; 1), (-1e+02; 0), (0; -1) 3. Area of figure: 1e+02 Select option: Rhombus: 1. Center: (3, 5) 2. Coordinates: (53; 5), (3; 5.5), (-47; 5), (3; 4.5) 3. Area of figure: 50 Rhombus: 1. Center: (0, 0) 2. Coordinates: (1e+02; 0), (0; 1), (-1e+02; 0), (0; -1) 3. Area of figure: 1e+02 Rhombus: 1. Center: (1, 1) 2. Coordinates: (21; 1), (1; 2), (-19; 1), (1; 0) 3. Area of figure: 21 Select option: Delete suboptions: 1. Delete rhombus from the beginning of the list 2. Delete rhombus from the end of the list 3. Delete rhombus from the list by index Select remove suboption: Rhombus successfully removed Select option: Delete suboptions: 1. Delete rhombus from the beginning of the list 2. Delete rhombus from the end of the list 3. Delete rhombus from the list by index Select remove suboption: Rhombus successfully removed Select option: Delete suboptions: 1. Delete rhombus from the beginning of the list 2. Delete rhombus from the end of the list 3. Delete rhombus from the list by index Select remove suboption: Rhombus successfully removed Select option: List is empty! Select option:

4. Листинг программы

main.cpp

```
/*
 *  Мариничев И. А.
 *  М8О-208Б-19
 *  github.com/IvaMarin/oop_exercise_06
 *  Вариант 13:
 *  Контейнер: список
 *  Фигура:    ромб
 *  Аллокатор: динамический массив
 */


#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. 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) { // 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 Allocator = std::allocator<T>>
class List {
private:
    struct Element;
    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;

        explicit 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=(List& other);
    T &operator[](size_t index);

private:
    using    allocator_type    =    typename    Allocator::template
rebind<Element>::other;

      struct deleter {
      private:
            allocator_type* allocator_;
      public:
            deleter(allocator_type*           allocator)           :
allocator_(allocator) {}

            void operator() (Element* ptr) {
                  if (ptr != nullptr) {

      std::allocator_traits<allocator_type>::destroy(*allocator_,
ptr);
                        allocator_->deallocate(ptr, 1);
                  }
            }

      };
    using unique_ptr = std::unique_ptr<Element, deleter>;
    struct Element {
        T value;
        unique_ptr next_element = { nullptr, deleter{nullptr} };
            Element* prev_element = nullptr;
            Element(const T& value_) : value(value_) {}
            forward_iterator Next();
    };

    allocator_type allocator_{};
      unique_ptr first{ nullptr, deleter{nullptr} };
    Element *tail = nullptr;
};

template<class T, class Allocator>
typename      List<T,      Allocator>::forward_iterator      List<T,
Allocator>::begin() {
    return forward_iterator(first.get());
}

template<class T, class Allocator>
typename      List<T,      Allocator>::forward_iterator      List<T,
Allocator>::end() {
```

```
    return forward_iterator(nullptr);
}

template<class T, class Allocator>
size_t List<T, Allocator>::Length() {
    return size;
}

template<class T, class Allocator>
bool List<T, Allocator>::Empty() {
    return Length() == 0;
}

template<class T, class Allocator>
void List<T, Allocator>::PushBack(const T& value) {
      Element* result = this->allocator_.allocate(1);
      std::allocator_traits<allocator_type>::construct(this-
>allocator_, result, value);
      if (!size) {
            first = unique_ptr(result, deleter{ &this->allocator_ });
            tail = first.get();
            size++;
            return;
      }
      tail->next_element   =   unique_ptr(result,   deleter{   &this-
>allocator_ });
      Element* temp =  tail;
      tail = tail->next_element.get();
      tail->prev_element = temp;
      size++;
}

template<class T, class Allocator>
void List<T, Allocator>::PushFront(const T& value) {
      size++;
      Element* result = this->allocator_.allocate(1);
      std::allocator_traits<allocator_type>::construct(this-
>allocator_, result, value);
      unique_ptr tmp = std::move(first);
      first = unique_ptr(result, deleter{ &this->allocator_ });
      first->next_element = std::move(tmp);
      if(first->next_element != nullptr)
            first->next_element->prev_element = first.get();
      if (size == 1) {
            tail = first.get();
      }
      if (size == 2) {
            tail = first->next_element.get();
      }
}

template<class T, class Allocator>
void List<T, Allocator>::PopFront() {
      if (size == 1) {
            first = nullptr;
            tail = nullptr;
            size--;
            return;
      }
      unique_ptr tmp = std::move(first->next_element);
      first = std::move(tmp);
```

```
	first->prev_element = nullptr;
	size--;
}

template<class T, class Allocator>
void List<T, Allocator>::PopBack() {
	if (tail->prev_element){
		Element* tmp = tail->prev_element;
		tail->prev_element->next_element = nullptr;
		tail = tmp;
	}
	else{
		first = nullptr;
		tail = nullptr;
	}
	size--;
}

template<class T, class Allocator>
T& List<T, Allocator>::Front() {
	if (size == 0) {
		throw std::logic_error("error: list is empty");
	}
	return first->value;
}

template<class T, class Allocator>
T& List<T, Allocator>::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, class allocator>
List<T,allocator>& List<T, allocator>::operator=(List<T, allocator>&
other) {
	size = other.size;
	first = std::move(other.first);
}

template<class T, class Allocator>
void		List<T,		Allocator>::EraseByIterator(List<T,
Allocator>::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;
	unique_ptr temp1 = std::move(d_it.it_ptr->next_element);
	d_it.it_ptr->prev_element->next_element = std::move(temp1);
	d_it.it_ptr = d_it.it_ptr->prev_element;
	d_it.it_ptr->next_element->prev_element = temp;

	size--;
}



template<class T, class Allocator>
void List<T, Allocator>::EraseByNumber(size_t N) {
	forward_iterator it = this->begin();
	for (size_t i = 0; i < N; ++i) {
		++it;
	}
	this->EraseByIterator(it);
}

template<class T, class Allocator>
void		List<T,		Allocator>::InsertByIterator(List<T,
Allocator>::forward_iterator ins_it, T& value) {
	Element* tmp = this->allocator_.allocate(1);
	std::allocator_traits<allocator_type>::construct(this-
>allocator_, tmp, 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;
	tmp->next_element = unique_ptr(ins_it.it_ptr, deleter{ &this-
>allocator_ });
	tmp->prev_element->next_element = unique_ptr(tmp, deleter{
&this->allocator_ });

	size++;
}

template<class T, class Allocator>
void List<T, Allocator>::InsertByNumber(size_t N, T& value) {
	forward_iterator it = this->begin();
	if (N >= this->Length())
		it = this->end();
	else
		for (size_t i = 0; i < N; ++i) {
			++it;
		}
	this->InsertByIterator(it, value);
}

template<class T, class allocator>
typename		List<T,allocator>::forward_iterator		List<T,
```

```
allocator>::Element::Next() {
      return forward_iterator(this->next_element.get());
}

template<class T, class Allocator>
List<T,        Allocator>::forward_iterator::forward_iterator(List<T,
Allocator>::Element *ptr) {
      it_ptr = ptr;
}

template<class T, class Allocator>
T& List<T, Allocator>::forward_iterator::operator*() {
      return this->it_ptr->value;
}


template<class T, class Allocator>
T& List<T, Allocator>::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, class Allocator>
typename      List<T,      Allocator>::forward_iterator&      List<T,
Allocator>::forward_iterator::operator++() {
      if (it_ptr == nullptr) {
        throw std::logic_error("error: out of range");
    }
      *this = it_ptr->Next();
      return *this;
}

template<class T, class Allocator>
typename      List<T,      Allocator>::forward_iterator      List<T,
Allocator>::forward_iterator::operator++(int) {
      forward_iterator old = *this;
      ++*this;
      return old;
}

template<class T, class Allocator>
bool      List<T,      Allocator>::forward_iterator::operator==(const
forward_iterator& other) const {
      return it_ptr == other.it_ptr;
}

template<class T, class Allocator>
bool      List<T,      Allocator>::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 */
```

vector.hpp

```
#ifndef VECTOR_HPP
#define VECTOR_HPP

#include <iostream>
#include <iterator>
#include <exception>
#include <memory>
#include <utility>
#include <algorithm>
#include <string>

template<typename T>
class Vector {
public:
      using value_type = T;
      using size_type = size_t;
      using difference_type = ptrdiff_t;
      using reference = value_type &;
      using const_reference = const value_type &;
      using pointer = value_type *;
      using const_pointer = const value_type *;
```

```
	class Iterator {
	public:
		using value_type = T;
		using difference_type = ptrdiff_t;
		using pointer = value_type *;
		using reference = value_type &;
		using 			iterator_category 			=
std::random_access_iterator_tag;

		Iterator(value_type *it = nullptr) : ptr{it} {}

		Iterator(const Iterator &other) : ptr{other.ptr} {}

		Iterator &operator=(const Iterator &other) {
			ptr = other.ptr;
		}

		Iterator operator--() {
			ptr--;
			return *this;
		}

		Iterator operator--(int s) {
			Iterator it = *this;
			--(*this);
			return it;
		}

		Iterator operator++() {
			ptr++;
			return *this;
		}

		Iterator operator++(int s) {
			Iterator it = *this;
			++(*this);
			return it;
		}

		reference operator*() {
			return *ptr;
		}

		pointer operator->() {
			return ptr;
		}

		bool operator==(const Iterator rhs) const {
			return ptr == rhs.ptr;
		}

		bool operator!=(const Iterator rhs) const {
			return ptr != rhs.ptr;
		}

		reference operator[](difference_type n) {
			return *(*this + n);
		}

		template<typename U>
```

```
            friend U &operator+=(U &r, typename U::difference_type n);

            template<typename U>
            friend U operator+(U a, typename U::difference_type n);

            template<typename U>
            friend U operator+(typename U::difference_type, U a);

            template<typename U>
            friend U &operator-=(U &r, typename U::difference_type n);

            template<typename U>
            friend typename U::difference_type operator-(U b, U a);

            template<typename U>
            friend bool operator<(U a, U b);

            template<typename U>
            friend bool operator>(U a, U b);

            template<typename U>
            friend bool operator==(U a, U b);

            template<typename U>
            friend bool operator>=(U a, U b);

            template<typename U>
            friend bool operator<=(U a, U b);

      private:
            value_type *ptr;
      };

      using iterator = Iterator;
      using const_iterator = const Iterator;


      Vector()    :    storageSize{0},    alreadyUsed{0},    storage{new
value_type[1]} {}

      Vector(size_t size) {
            if (size < 0) {
                  throw std::logic_error("error: size must be >= 0");
            }
            alreadyUsed = 0;
            storageSize = size;
            storage = new value_type[size + 1];
      }

      ~Vector() {
            alreadyUsed = storageSize = 0;
            delete [] storage;
            storage = nullptr;
      }

      size_t Size() const {
            return alreadyUsed;
      }

      bool Empty() const {
            return Size() == 0;
```

```
	}

	iterator Begin() {
		if (!Size())
			return nullptr;
		return storage;
	}

	iterator End() {
		if (!Size())
			return nullptr;
		return (storage + alreadyUsed);
	}

	const_iterator Begin() const {
		if (!Size())
			return nullptr;
		return storage;
	}

	const_iterator End() const {
		if (!Size())
			return nullptr;
		return (storage + alreadyUsed);
	}

	reference Front() {
		return storage[0];
	}

	const_reference Front() const {
		return storage[0];
	}

	reference Back() {
		return storage[alreadyUsed - 1];
	}

	const_reference Back() const {
		return storage[alreadyUsed - 1];
	}

	reference At(size_t index) {
		if (index < 0 || index >= alreadyUsed) {
			throw std::out_of_range("error: the index must be
greater than or equal to zero and less than the number of elements");
		}

		return storage[index];
	}

	const_reference At(size_t index) const {
		if (index < 0 || index >= alreadyUsed) {
			throw std::out_of_range("error: the index must be
greater than or equal to zero and less than the number of elements");
		}

		return storage[index];
	}

	reference operator[](size_t index) {
```

```
            return storage[index];
      }

      const_reference operator[](size_t index) const {
            return storage[index];
      }

      size_t getStorageSize() const {
            return storageSize;
      }

      void PushBack(const T& value) {
            if (alreadyUsed < storageSize) {
                  storage[alreadyUsed] = value;
                  ++alreadyUsed;
                  return;
            }

            size_t nextSize = 1;
            if (!Empty()) {
                  nextSize = storageSize * 2;
            }

            Vector<T> next{nextSize};
            next.alreadyUsed = alreadyUsed;
            std::copy(Begin(), End(), next.Begin());
            next[alreadyUsed] = value;
            ++next.alreadyUsed;
            Swap(*this, next);
      }

      void PopBack() {
            if (alreadyUsed) {
                  alreadyUsed--;
            }
      }

      iterator Erase(const_iterator pos) {
            Vector<T> newVec{getStorageSize()};
            Iterator newIt = newVec.Begin();
            for (Iterator it = Begin(); it != pos; it++, newIt++) {
                  *newIt = *it;
            }
            Iterator result = newIt;
            for (Iterator it = pos + 1; it != End(); it++, newIt++) {
                  *newIt = *it;
            }
            newVec.alreadyUsed = alreadyUsed - 1;
            Swap(*this, newVec);

            return result;
      }


      template<typename U>
      friend void Swap(Vector<U> &lhs, Vector<U> &rhs);

private:
      size_t storageSize;
      size_t alreadyUsed;
      value_type *storage;
```

```
};

template<typename T>
T &operator+=(T &r, typename T::difference_type n) {
      r.ptr = r.ptr + n;
      return r;
}

template<typename T>
T operator+(T a, typename T::difference_type n) {
      T temp = a;
      temp += n;
      return temp;
}

template<typename T>
T operator+(typename T::difference_type n, T a) {
      return a + n;
}

template<typename T>
T &operator-=(T &r, typename T::difference_type n) {
      r.ptr = r.ptr - n;
      return r;
}

template<typename T>
typename T::difference_type operator-(T b, T a) {
      return b.ptr - a.ptr;
}

template<typename T>
bool operator<(T a, T b) {
      return a - b < 0 ? true : false;
}

template<typename T>
bool operator>(T a, T b) {
      return b < a;
}

template<typename T>
bool operator==(T a, T b) {
      return a - b == 0 ? true : false;
}

template<typename T>
bool operator>=(T a, T b) {
      return a > b || a == b;
}

template<typename T>
bool operator<=(T a, T b) {
      return a < b || a == b;
}

template<typename U>
void Swap(Vector<U> &lhs, Vector<U> &rhs) {
      std::swap(lhs.alreadyUsed, rhs.alreadyUsed);
      std::swap(lhs.storageSize, rhs.storageSize);
      std::swap(lhs.storage, rhs.storage);
```

```
}


#endif /* VECTOR_HPP */
```

allocator.hpp

```
#ifndef ALLOCATOR_HPP
#define ALLOCATOR_HPP

#include <iostream>
#include <exception>
#include "vector.hpp"

template<typename T, size_t ALLOC_SIZE>
class Allocator {
public:
      using value_type = T;
      using size_type = size_t;
      using difference_type = ptrdiff_t;
      using is_always_equal = std::false_type;

      template<typename U>
      struct rebind {
            using other = Allocator<U, ALLOC_SIZE>;
      };

      Allocator() : begin{new char[ALLOC_SIZE]},
      end{begin + ALLOC_SIZE}, tail{begin} {}

      Allocator(const Allocator&) = delete;
      Allocator(Allocator &&) = delete;

      ~Allocator() {
            delete [] begin;
            begin = end = tail = nullptr;
            freeBlocks.~Vector();
      }

      T *Allocate(size_t n) {
            if (n != 1) {
                  throw std::logic_error("error: This allocator can't
allocate arrays");
            }
            if (end - tail < sizeof(T)) {
                  if (!freeBlocks.Empty()) {
                        char *ptr = freeBlocks.Back();
                        freeBlocks.PopBack();
                        return reinterpret_cast<T *>(ptr);
                  }
                  throw std::bad_alloc();
            }
            T *result = reinterpret_cast<T *>(tail);
            tail += sizeof(T);
            return result;
      }

      void deallocate(T *ptr, size_t n) {
            if (n != 1) {
                  throw std::logic_error("error: This allocator can't
deallocate arrays");
            }
```

```
            if (ptr == nullptr) {
                  return;
            }
            freeBlocks.PushBack(reinterpret_cast<char *>(ptr));
      }

private:
      char *begin;
      char *end;
      char *tail;
      Vector<char *> freeBlocks;
};

#endif /* ALLOCATOR_HPP */
```

CmakeLists.txt

```
cmake_minimum_required(VERSION 3.17)
project(oop_lab6)

set(CMAKE_CXX_STANDARD 17)

add_executable(oop_lab6 main.cpp)
```

5. Вывод

   В ходе лабораторной работы я изучил основы работы с контейнерами в C++, познакомился с концепцией аллокаторов памяти.

Список литературы

1. Стефан К. Дьюхэрст Скользкие места C++. Как избежать проблем при проектировании и компиляции ваших программ — ISBN: 5-94074-083-9 — 266 с.
2. Forward Iterators in C++ [Электронный ресурс]. URL: https://www.geeksforgeeks.org/forward-iterators-in-cpp/ (дата обращения: 4.12.2020).
3. auto_ptr, unique_ptr, shared_ptr and weak_ptr [Электронный ресурс]. URL: https://www.geeksforgeeks.org/auto_ptr-unique_ptr-shared_ptr-weak_ptr-2/ (дата обращения: 5.12.2020).
4. std::list [Электронный ресурс]. URL: https://www.cplusplus.com/reference/list/list/ (дата обращения: 5.12.2020).
5. Area of a rhombus [Электронный ресурс]. URL: https://www.mathopenref.com/rhombusarea.html (дата обращения: 5.12.2020).