Московский Авиационный Институт
(Национальный исследовательский Университет)

Факультет: «Информационные технологии и прикладная математика»
Кафедра: 806 «Вычислительная математика и программирование»

**Лабораторная работа**
**по курсу «ООП»**

**Тема:**
**«Основы работы с коллекциями: итераторы»**

| | |
|---|---|
| Студент: | Ли А. И. |
| Группа: | М80-208Б-18 |
| Преподаватель: | Журавлев А.А. |
| Вариант: | 13 |
| Оценка: | |
| Дата: | |

Москва
2019

## 1. Задание

Разработать шаблоны классов согласно варианту задания. Параметром шаблона должен являться скалярный тип данных задающий тип данных для оси координат. Классы должны иметь публичные поля. Фигуры являются фигурами вращения. Для хранения координат фигур необходимо использовать шаблон std::pair.

Создать шаблон динамической коллекцию, согласно варианту (13: треугольник, список):

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. 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 аз границы коллекции или удаление не

существующего 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 [];

10. 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);

## 2. Адрес репозитория на GitHub

https://github.com/elips0n/oop_exercise_05

## 3. Код программы на C++

main.cpp

```
#include <algorithm>
#include <iostream>
#include "triangle.h"
#include "TList.h"

void menu()
{
    std::cout << " \n Выберите действие:" << std::endl;
    std::cout << "1) Добавить треугольник в список" << std::endl;
    std::cout << "2) Удалить треугольник из списка" << std::endl;
    std::cout << "3) Вывести количество элементов, площадь которых меньше заданной (std::count_if)" << std::endl;
    std::cout << "4) Печать списка фигур с помощью std::for_each()" << std::endl;
    std::cout << "0) Выход" << std::endl;
}
```

```
float param;
bool comp(std::shared_ptr<TListItem<Triangle>> i) { // функция сравнения для
count_if
    if ((float)(i.get()->GetFigure()->Square()) < param) { return true; }
    else return false;
}

uint fc;
void output(std::shared_ptr<TListItem<Triangle>> i) { // функция для цикла
for_each
    std::cout << "\n Фигура № " << fc << std::endl;
    i.get()->GetFigure()->Print();
    fc++;
}

int main(void)
{
   int32_t act = 0;
   TList<Triangle> list;
   std::shared_ptr<Triangle> ptr;
        do {
                menu();
                std::cin >> act;
                switch (act) {
                case 1:
                        ptr = std::make_shared<Triangle>(std::cin);
                        list.Insert(ptr);
                        break;
                case 2:
                        list.Erase();
                        break;
                case 3:
                        if (!list.IsEmpty()) {
                                std::cout << "Введите величину максимальной
площади\n" << std::endl;
                                std::cin >> param;
                                std::cout << "Количество элементов с площадью
меньше заданной: ";
                                std::cout << std::count_if(list.begin(), list.end(), comp);
//подсчет с помощью  count_if
                                std::cout << std::endl << "------------------\n" << std::endl;
                        }
                        else {
                                std::cout << "В списке нет фигур." << std::endl;
```

```
                }
                break;
            case 4:
                if (!list.IsEmpty()) {
                    fc = 0;
                    std::for_each(list.begin(), list.end(), output); //вывод с
помощью  for_each
                }
                else {
                    std::cout << "В списке нет фигур." << std::endl;
                }
                break;
            case 0:
                list.Del();
                break;
            default:
                std::cout << "Неопознанная команда." << std::endl;;
                break;
            }
        } while (act);
    return 0;
}
```

Iterator.h

```
#ifndef ITERATOR_H
#define ITERATOR_H

#include <memory>
#include <iostream>
#include "TListItem.h"

template <class N, class T>
class forward_iterator
{
public:
    using value_type = T;
    using reference = T & ;
    using pointer = T * ;
    using difference_type = ptrdiff_t;
    using iterator_category = std::forward_iterator_tag;

    forward_iterator(std::shared_ptr<N> n) {
      cur = n;
    }
```

```
    std::shared_ptr<N> operator* () {
      return cur;
    }

    std::shared_ptr<T> operator-> () {
      return cur->GetFigure();
    }

    void operator++() {
        if (((!cur)&&(!(cur->GetNext())))) {
                throw std::logic_error("попытка доступа к несуществующему
элементу");
        }
      cur = cur->GetNext();
    }

     forward_iterator operator++ (int) {
        forward_iterator cur(*this);
      ++(*this);
      return cur;
    }

    void operator--() {
        if (((!cur) && (!(cur->GetPrev())))) {
                throw std::logic_error("попытка доступа к несуществующему
элементу");
        }
      cur = cur->GetPrev();
    }

     forward_iterator operator-- (int) {
        forward_iterator cur(*this);
      --(*this);
      return cur;
    }

    bool operator== (const forward_iterator &i) {
      return (cur == i.cur);
    }

    bool operator!= (const forward_iterator &i) {
      return (cur != i.cur);
    }
```

```
private:
   std::shared_ptr<N> cur;
};

#endif
```

list.h

```
#pragma once
#include <iterator>
#include <memory>


namespace containers {

    template<class T, class Allocator = std::allocator<T>>
    class list {
    private:
        struct element;//объявление типа хранящегося в list, для того, чтобы он был виден forward_iterator
        size_t size = 0;//размер списка
    public:
        list() = default;//коструктор по умолчанию

        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 push_back(const T& value);
```

```
        void push_front(const T& value);
        T& front();
        T& back();
        void pop_back();
        void pop_front();
        size_t length();
        bool empty();
        void delete_by_it(forward_iterator d_it);
        void delete_by_number(size_t N);
        void insert_by_it(forward_iterator ins_it, T& value);
        void insert_by_number(size_t N, T& value);
        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>::push_back(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>::push_front(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>::pop_front() {
    if (size == 0) {
            throw std::logic_error("can`t pop from empty list");
    }
    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>::pop_back() {
    if (size == 0) {
            throw std::logic_error("can`t pop from empty list");
    }
    if (tail->prev_element){
            element* tmp = tail->prev_element;
            tail->prev_element->next_element = nullptr;
    }
    else{
            first = nullptr;
            tail = nullptr;
    }
    size--;
}
```

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

template<class T, class Allocator>
T& list<T, Allocator>::back() {
    if (size == 0) {
        throw std::logic_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>::delete_by_it(containers::list<T,
Allocator>::forward_iterator d_it) {
    forward_iterator i = this->begin(), end = this->end();
    if (d_it == end) throw std::logic_error("out of borders");
    if (d_it == this->begin()) {
        this->pop_front();
        return;
    }
    if (d_it.it_ptr == tail) {
        this->pop_back();
        return;
    }

    if (d_it.it_ptr == nullptr) throw std::logic_error("out of broders");
    auto temp = d_it.it_ptr->prev_element;
    unique_ptr 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, class Allocator>
    void list<T, Allocator>::delete_by_number(size_t N) {

        if (this->length() == 0)
        {
                std::cerr << "Нет фигур для удаления. Длина списка 0.\n\n";
                return;
        }
        if (N<0 || N>(this->length())-1)
        {
                std::cerr << "Введенный индекс находиться за пределами
возможных значений\n\n";
                return;
        }
        if (N==(this->length()) - 1)
        {
                pop_back();
                std::cout << "Фигура удалена из списка.\n" << std::endl;
                return;
        }
        forward_iterator it = this->begin();
        for (size_t i = 0; i < N; ++i) {
                ++it;
        }
        this->delete_by_it(it);
        std::cout << "Фигура удалена из списка.\n" << std::endl;
    }

    template<class T, class Allocator>
    void list<T, Allocator>::insert_by_it(containers::list<T,
Allocator>::forward_iterator ins_it, T& value) {

        if (ins_it == this->begin()) {
                this->push_front(value);
                return;
        }
        if(ins_it.it_ptr == nullptr){
                this->push_back(value);
                return;
        }
        element* tmp = this->allocator_.allocate(1);
```

```
        std::allocator_traits<allocator_type>::construct(this->allocator_, tmp, value);

        forward_iterator i = this->begin();

        tmp->prev_element = ins_it.it_ptr->prev_element;
        ins_it.it_ptr->prev_element = tmp;
        tmp->next_element = std::move(tmp->prev_element->next_element);
        tmp->prev_element->next_element = unique_ptr(tmp, deleter{ &this->allocator_ });

        size++;
    }

    template<class T, class Allocator>
    void list<T, Allocator>::insert_by_number(size_t N, T& value) {
        if (N<0 || N>this->length())
        {
                std::cerr << "Введенный индекс находиться за пределами возможных значений\n\n";
                return;
        }
        if (N==0)
        {
                push_front(value);
                return;
        }

        forward_iterator it = this->begin();
        for (size_t i = 0; i < N; ++i) {
                ++it;
        }
        this->insert_by_it(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(containers::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::out_of_range("out of list's borders");
        }
        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("out of list borders");
        *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;
    }
}
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)
project(oop5)

set(CMAKE_CXX_STANDARD 17)

add_executable(main main.cpp TList.cpp TListItem.cpp triangle.cpp )
```

## 4. Вывод

В ходе данной лабораторной работы мною были получены навыки работы с основами коллекций, а конкретно с итераторами. Благодаря итераторам, при их грамотной настройке программист получает более наглядный и простой способ работы с контейнерами и другими абстрактными типами данных, кроме того, правильная реализация итераторов в собственном типе данных дает программисту возможность использования уже написанных алгоритмов, в основе которых лежит взаимодействие через итераторы.