Московский Авиационный Институт

(Национальный Исследовательский Университет)

Кафедра 806 «Вычислительная информатика и программирование»

Факультет: «Информационные технологии и прикладная математика»

Лабораторная работа

Дисциплина: «Объектно-ориентированное программирование»

III семестр

Задание 5: «Основные работы с коллекциями: итераторы»

|  |  |
| --- | --- |
| Группа: | М8О-208Б-18, №9 |
| Студент: | Игитова Александра Андреевна |
| Преподаватель: | Журавлёв Андрей Андреевич |
| Оценка: |  |
| Дата: | 11.01.2020 |

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 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;

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 c помощью std::for\_each;

Выводит на экран количество объектов, у которых площадь меньше заданной (с помощью std::count\_if);

*Вариант 9: треугольник, список*

1. **Адрес репозитория на GitHub**

<https://github.com/SandraIgitova/oop_exercise_05>

1. **Код программы на С++**

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 )

1. **Вывод**

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. ☺