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 6: «Основные работы с коллекциями: итераторы»

|  |  |
| --- | --- |
| Группа: | М8О-206Б-18, №14 |
| Студент: | Орозбакиев Э.Д. |
| Преподаватель: | Журавлёв Андрей Андреевич |
| Оценка: |  |
| Дата: |  |

Москва, 2019

1. **Задание**

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.

Необходимо настроить сборку лабораторной работы с помощью CMake. Собранная программа должна называться oop\_exercise\_06 (в случае использования Windows oop\_exercise\_06.exe)

Необходимо зарегистрироваться на GitHub (если студент уже имеет регистрацию на GitHub то можно использовать ее) и создать репозитарий для задания лабораторной работы.

Преподавателю необходимо предъявить ссылку на публичный репозиторий на Github. Имя репозитория должно быть <https://github.com/login/oop_exercise_06>

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. Коллекция должна содержать метод доступа:

Стек – pop, push, top;

Очередь – pop, push, top;

Список,Динамический массив – доступ к элементу по оператору [];

Реализовать аллокатор, 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 (Динамический массив, Список, Стек, Очередь);

Коллекция должна использовать аллокатор для выделеления и освобождения памяти для своих элементов.

Аллокатор должен быть совместим с контейнерами std::map и std::list (опционально – vector).

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 c помощью std::for\_each;

1. **Адрес репозитория на GitHub**

https://github.com/p0kemo4ik/oop\_exercise\_06

1. **Код программы на С++**

*main.cpp*

#include **<iostream>**  
#include **<algorithm>**  
  
#include **"list.hpp"**  
#include **"allocator.hpp"**  
#include **"pentagon.hpp"**  
  
**int** main() {  
 container::list<Pentagon<**double**>, allocator::my\_allocator<Pentagon<**double**>, 500>> list;  
 **int** command, pos;  
  
 **while**(**true**) {  
 std::cout << std::endl;  
 std::cout << **"1. Добавить фигуру в список"**<< std::endl;  
 std::cout << **"2. Удалить фигуру"** << std::endl;  
 std::cout << **"3. Вывести все фигуры"** << std::endl;  
 std::cout << **"4. Вывести кол-во фигур чья площаль больше чем ..."** << std::endl;  
 std::cout << **"5. Вывести фигуру"** << std::endl << std::endl;  
 std::cin >> command;  
  
 **if** (command == 0) {  
 **break**;  
  
 } **else if**(command == 1) {  
 std::cout << **"Введите координаты"** << std::endl;  
 Pentagon<**double**> pentagon(std::cin);  
  
 std::cout << **"1. Добавить фигуру в начало списка"** << std::endl;  
 std::cout << **"2. Добавить фигуру по индексу"** << std::endl;  
 std::cin >> command;  
 **if**(command == 1) {  
 list.push(pentagon);  
 **continue**;  
 } **else if**(command == 2) {  
 std::cout << **"Введите индекс"** << std::endl;  
 std::cin >> pos;  
 list.insert(pos, pentagon);  
 **continue**;  
 } **else** {  
 std::cout << **"Неправильная команда"** << std::endl;  
 std::cin >> command;  
 **continue**;  
 }  
  
 } **else if**(command == 2) {  
 std::cout << **"1. Удалить фигуру из списка по индексу"** << std::endl;  
 std::cout << **"2. Удалить по итератору"** << std::endl;  
 std::cout << **"3. Удалить фигуру из начала списка"** << std::endl;  
 std::cin >> command;  
 **if** (command == 1) {  
 std::cout << **"Введите индекс"** << std::endl;  
 std::cin >> pos;  
 list.erase(pos);  
 **continue**;  
 } **else if** (command == 2) {  
 std::cout << **"Введите индекс"** << std::endl;  
 std::cin >> pos;  
 **auto** temp = list.begin();  
 **for**(**int** i = 0; i < pos; ++i) {  
 ++temp;  
 }  
 list.erase(temp);  
 **continue**;  
  
 } **else if** (command == 3) {  
 **try** {  
 list.popFront();  
 } **catch**(std::exception& e) {  
 std::cout << e.what() << std::endl;  
 **continue**;  
 }  
 } **else** {  
 std::cout << **"Неправильная команда"** << std::endl;  
 std::cin >> command;  
 **continue**;  
 }  
 } **else if**(command == 3) {  
 **for**(**const auto**& item : list) {  
 item.print(std::cout);  
 std::cout << **"Center: ["** << item.center() << **"]"** << std::endl;  
 std::cout << **"Area: "** << item.square() << std::endl;  
 **continue**;  
 }  
  
 } **else if**(command == 4) {  
 std::cout << **"Введите площадь"** << std::endl;  
 std::cin >> pos;  
 std::cout << **"Количество пятиугольнков площадь, которых меньше заданной "** << pos;  
 std::cout << std::count\_if(list.begin(), list.end(), [pos](Pentagon<**double**> square) {**return** square.square() < pos;}) << std::endl;  
 **continue**;  
  
 } **else if** (command == 5) {  
 std::cout << **"Введите номер элемента"** << std::endl;  
 std::cin >> pos;  
 **try** {  
 list[pos].print(std::cout);  
 std::cout << **"Center: ["** << list[pos].center() << **"]"** << std::endl;  
 std::cout << **"Area: "** << list[pos].square() << std::endl;  
 } **catch**(std::exception& e) {  
 std::cout << e.what() << std::endl;  
 **continue**;  
 }  
 **continue**;  
  
 } **else** {  
 std::cout << **"Неправильная команда"** << std::endl;  
 **continue**;  
 }  
 }  
  
 **return** 0;  
}

*list.hpp*

#pragma once  
  
#include **<iterator>**  
#include **<memory>**  
#include **<iostream>**  
  
**namespace** container {  
  
 **template**<**class** T, **class** Allocator = std::allocator<T>>  
 **class** list {  
 **private**:  
 **struct** node\_t;  
 size\_t size = 0;  
  
 **public**:  
 **struct** forward\_iterator {  
 **using** value\_type = T;  
 **using** reference = T&;  
 **using** pointer = T\*;  
 **using** difference\_type = ptrdiff\_t;  
 **using** iterator\_category = std::forward\_iterator\_tag;  
  
 **explicit** forward\_iterator(node\_t\* ptr);  
 T& **operator**\*();  
 forward\_iterator& **operator**++();  
 forward\_iterator **operator**++(**int**);  
 **bool operator**==(**const** forward\_iterator& it) **const**;  
 **bool operator**!=(**const** forward\_iterator& it) **const**;  
 **private**:  
 node\_t\* ptr\_;  
 **friend** list;  
 };  
  
 forward\_iterator begin();  
 forward\_iterator **end**();  
 **void** push(**const** T& value);  
 **void** push\_b(**const** T& value);  
 T& front();  
 T& back();  
 **void** popFront();  
 **void** popBack();  
 size\_t length();  
 **bool** empty();  
 **void** erase(forward\_iterator d\_it);  
 **void** erase(size\_t N);  
 **void** insert\_by\_it(forward\_iterator ins\_it, T& value);  
 **void** insert(size\_t N, T& value);  
 list& **operator**=(list& other);  
 T& **operator**[](size\_t index);  
  
 **private**:  
 **using** allocator\_type = **typename** Allocator::**template** rebind<node\_t>::other;  
  
 **struct** deleter {  
 **private**:  
 allocator\_type\* allocator\_;  
 **public**:  
 deleter(allocator\_type\* allocator) : allocator\_(allocator) {}  
  
 **void operator**() (node\_t\* ptr) {  
 **if** (ptr != **nullptr**) {  
 std::allocator\_traits<allocator\_type>::destroy(\*allocator\_, ptr);  
 allocator\_->***deallocate***(ptr, 1);  
 }  
 }  
  
 };  
  
 **using** unique\_ptr = std::unique\_ptr<node\_t, deleter>;  
  
 **struct** node\_t {  
 T value;  
 unique\_ptr next\_element = { **nullptr**, deleter{**nullptr**} };  
 node\_t\* prev\_element = **nullptr**;  
 node\_t(**const** T& value\_) : value(value\_) {}  
 forward\_iterator next();  
 };  
  
 allocator\_type allocator\_{};  
 unique\_ptr head{ **nullptr**, deleter{**nullptr**} };  
 node\_t\* tail = **nullptr**;  
 };  
  
 **template**<**class** T, **class** Allocator>  
 **typename** list<T, Allocator>::forward\_iterator list<T, Allocator>::begin() {*//+*  
**return** forward\_iterator(head.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(**const** T& value) {  
 size++;  
 node\_t\* result = **this**->allocator\_.***allocate***(1);  
 std::allocator\_traits<allocator\_type>::construct(**this**->allocator\_, result, value);  
 unique\_ptr tmp = std::move(head);  
 head = unique\_ptr(result, deleter{ &**this**->allocator\_ });  
 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, **class** Allocator>  
 **void** list<T, Allocator>::push\_b(**const** T& value) {  
 node\_t\* result = **this**->allocator\_.allocate(1);  
 std::allocator\_traits<allocator\_type>::construct(**this**->allocator\_, result, value);  
 **if** (!size) {  
 head = unique\_ptr(result, deleter{ &**this**->allocator\_ });  
 tail = head.get();  
 size++;  
 **return**;  
 }  
 tail->next\_element = unique\_ptr(result, deleter{ &**this**->allocator\_ });  
 node\_t\* temp = tail;  
 tail = tail->next\_element.get();  
 tail->prev\_element = temp;  
 size++;  
 }  
  
  
 **template**<**class** T, **class** Allocator>  
 **void** list<T, Allocator>::popFront() {  
 **if** (size == 0) {  
 **throw** std::logic\_error(**"Deleting from empty list"**);  
 }  
 **if** (size == 1) {  
 head = **nullptr**;  
 tail = **nullptr**;  
 size--;  
 **return**;  
 }  
 unique\_ptr tmp = std::move(head->next\_element);  
 head = std::move(tmp);  
 head->prev\_element = **nullptr**;  
 size--;  
 }  
  
 **template**<**class** T, **class** Allocator>  
 **void** list<T, Allocator>::popBack() {  
 **if** (size == 0) {  
 **throw** std::logic\_error(**"Deleting from empty list"**);  
 }  
 **if** (tail->prev\_element){  
 node\_t\* tmp = tail->prev\_element;  
 tail->prev\_element->next\_element = **nullptr**;  
 tail = tmp;  
 }  
 **else**{  
 head = **nullptr**;  
 tail = **nullptr**;  
 }  
 size--;  
 }  
  
 **template**<**class** T, **class** Allocator>  
 T& list<T, Allocator>::front() {  
 **if** (size == 0) {  
 **throw** std::logic\_error(**"No elements"**);  
 }  
 **return** head->value;  
 }  
  
 **template**<**class** T, **class** Allocator>  
 list<T,Allocator>& list<T, Allocator>::**operator**=(list<T, Allocator>& other) {  
 size = other.size;  
 head = std::move(other.head);  
 }  
  
 **template**<**class** T, **class** Allocator>  
 **void** list<T, Allocator>::erase(container::list<T, Allocator>::forward\_iterator d\_it) {  
 **if** (d\_it == **this**->**end**()) **throw** std::logic\_error(**"Out of bounds"**);  
 **if** (d\_it == **this**->begin()) {  
 **this**->popFront();  
 **return**;  
 }  
 **if** (d\_it.ptr\_ == tail) {  
 **this**->popBack();  
 **return**;  
 }  
 **if** (d\_it.ptr\_ == **nullptr**) **throw** std::logic\_error(**"Out of bounds"**);  
 **auto** temp = d\_it.ptr\_->prev\_element;  
 unique\_ptr temp1 = std::move(d\_it.ptr\_->next\_element);  
 d\_it.ptr\_ = d\_it.ptr\_->prev\_element;  
 d\_it.ptr\_->next\_element = std::move(temp1);  
 d\_it.ptr\_->next\_element->prev\_element = temp;  
 size--;  
 }  
  
 **template**<**class** T, **class** Allocator>  
 **void** list<T, Allocator>::erase(size\_t N) {  
 forward\_iterator it = **this**->begin();  
 **for** (size\_t i = 0; i < N; ++i) {  
 ++it;  
 }  
 **this**->erase(it);  
 }  
  
 **template**<**class** T, **class** Allocator>  
 **void** list<T, Allocator>::insert\_by\_it(container::list<T, Allocator>::forward\_iterator ins\_it, T& value) {  
  
 **if** (ins\_it == **this**->begin()) {  
 **this**->push(value);  
 **return**;  
 }  
 **if**(ins\_it.ptr\_ == **nullptr**){  
 **this**->push\_b(value);  
 **return**;  
 }  
  
 node\_t\* tmp = **this**->allocator\_.allocate(1);  
 std::allocator\_traits<allocator\_type>::construct(**this**->allocator\_, tmp, value);  
  
 tmp->prev\_element = ins\_it.ptr\_->prev\_element;  
 ins\_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(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**->insert\_by\_it(it, value);  
 }  
  
 **template**<**class** T, **class** Allocator>  
 **typename** list<T,Allocator>::forward\_iterator list<T, Allocator>::node\_t::next() {  
 **return** forward\_iterator(**this**->next\_element.get());  
 }  
  
 **template**<**class** T, **class** Allocator>  
 list<T, Allocator>::forward\_iterator::forward\_iterator(container::list<T, Allocator>::node\_t \*ptr) {  
 ptr\_ = ptr;  
 }  
  
 **template**<**class** T, **class** Allocator>  
 T& list<T, Allocator>::forward\_iterator::**operator**\*() {  
 **return this**->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 bounds"**);  
 }  
 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** (ptr\_ == **nullptr**) **throw** std::logic\_error(**"Out of list bounds"**);  
 \***this** = 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** ptr\_ == other.ptr\_;  
 }  
  
 **template**<**class** T, **class** Allocator>  
 **bool** list<T, Allocator>::forward\_iterator::**operator**!=(**const** forward\_iterator& other) **const** {  
 **return** ptr\_ != other.ptr\_;  
 }  
}

*allocator.h*

#ifndef **D\_ALLOCATOR\_H\_**#define **D\_ALLOCATOR\_H\_** 1#include **<cstdlib>**#include **<iostream>**#include **<type\_traits>**#include **<list>**#include **"list.hpp"****namespace** allocator { **template**<**class** T, size\_t ***ALLOC\_SIZE***> **struct** my\_allocator { **using** value\_type = T; **using** size\_type = std::size\_t; **using** difference\_type = std::ptrdiff\_t; **using** is\_always\_equal = std::false\_type; **template**<**class** L> **struct** rebind { **using** other = my\_allocator<L, ***ALLOC\_SIZE***>; }; my\_allocator() : pool\_begin(**new char**[***ALLOC\_SIZE***]), pool\_end(pool\_begin + ***ALLOC\_SIZE***), pool\_tail(pool\_begin) {} my\_allocator(**const** my\_allocator&) = **delete**; my\_allocator(my\_allocator&&) = **delete**; ~my\_allocator() { **delete**[] pool\_begin; } T\* allocate(std::size\_t n); **void** deallocate(T\* ptr, std::size\_t n); **private**: **char**\* pool\_begin; **char**\* pool\_end; **char**\* pool\_tail; std::list<**char**\*> free\_blocks; }; **template**<**class** T, size\_t ***ALLOC\_SIZE***> T\* my\_allocator<T, ***ALLOC\_SIZE***>::allocate(std::size\_t n) { **if** (n != 1) { **throw** std::logic\_error(**"Allocating arrays is unavaliable"**); } **if** (size\_t(pool\_end - pool\_tail) < **sizeof**(T)) { **if** (free\_blocks.size()) { **auto** it = free\_blocks.begin(); **char**\* ptr = \*it; free\_blocks.pop\_front(); **return reinterpret\_cast**<T\*>(ptr); } **throw** std::bad\_alloc(); } T\* result = **reinterpret\_cast**<T\*>(pool\_tail); pool\_tail += **sizeof**(T); **return** result; } **template**<**class** T, size\_t ***ALLOC\_SIZE***> **void** my\_allocator<T, ***ALLOC\_SIZE***>::deallocate(T\* ptr, std::size\_t n) { **if** (n != 1) { **throw** std::logic\_error(**"Allocating arrays is unavaliable, thus deallocating is unavalivable as well"**); } **if** (ptr == **nullptr**) { **return**; } free\_blocks.push\_back(**reinterpret\_cast**<**char**\*>(ptr)); }};#endif *// D\_ALLOCATOR\_H\_*

1. **Результаты выполнения тестов**

1. Добавить фигуру в список

2. Удалить фигуру

3. Вывести все фигуры

4. Вывести кол-во фигур чья площаль больше чем ...

5. Вывести фигуру

1

Введите координаты

0 0 0 0 0 0 0 0 0 0

1. Добавить фигуру в начало списка

2. Добавить фигуру по индексу

1

1. Добавить фигуру в список

2. Удалить фигуру

3. Вывести все фигуры

4. Вывести кол-во фигур чья площаль больше чем ...

5. Вывести фигуру

1

Введите координаты

1 1 1 1 1 1 1 1 1 1

1. Добавить фигуру в начало списка

2. Добавить фигуру по индексу

2

Введите индекс

0

1. Добавить фигуру в список

2. Удалить фигуру

3. Вывести все фигуры

4. Вывести кол-во фигур чья площаль больше чем ...

5. Вывести фигуру

1

Введите координаты

2 2 2 2 2 2 2 2 2 2

1. Добавить фигуру в начало списка

2. Добавить фигуру по индексу

2

Введите индекс

1

1. Добавить фигуру в список

2. Удалить фигуру

3. Вывести все фигуры

4. Вывести кол-во фигур чья площаль больше чем ...

5. Вывести фигуру

1

Введите координаты

3 3 3 3 3 3 3 3 3 3

1. Добавить фигуру в начало списка

2. Добавить фигуру по индексу

1

1. Добавить фигуру в список

2. Удалить фигуру

3. Вывести все фигуры

4. Вывести кол-во фигур чья площаль больше чем ...

5. Вывести фигуру

3

Pentagon: [3, 3] [3, 3] [3, 3] [3, 3] [3, 3]

Center: [[3, 3]]

Area: 0

Pentagon: [1, 1] [1, 1] [1, 1] [1, 1] [1, 1]

Center: [[1, 1]]

Area: 0

Pentagon: [2, 2] [2, 2] [2, 2] [2, 2] [2, 2]

Center: [[2, 2]]

Area: 0

Pentagon: [0, 0] [0, 0] [0, 0] [0, 0] [0, 0]

Center: [[0, 0]]

Area: 0

1. Добавить фигуру в список

2. Удалить фигуру

3. Вывести все фигуры

4. Вывести кол-во фигур чья площаль больше чем ...

5. Вывести фигуру

2

1. Удалить фигуру из списка по индексу

2. Удалить по итератору

3. Удалить фигуру из начала списка

3

1. Добавить фигуру в список

2. Удалить фигуру

3. Вывести все фигуры

4. Вывести кол-во фигур чья площаль больше чем ...

5. Вывести фигуру

3

Pentagon: [1, 1] [1, 1] [1, 1] [1, 1] [1, 1]

Center: [[1, 1]]

Area: 0

Pentagon: [2, 2] [2, 2] [2, 2] [2, 2] [2, 2]

Center: [[2, 2]]

Area: 0

Pentagon: [0, 0] [0, 0] [0, 0] [0, 0] [0, 0]

Center: [[0, 0]]

Area: 0

1. Добавить фигуру в список

2. Удалить фигуру

3. Вывести все фигуры

4. Вывести кол-во фигур чья площаль больше чем ...

5. Вывести фигуру

0

1. **Объяснение результатов работы программы**

Программа выводит меню, в котором описываются все применимые к фигурам функции – вставка, удаление и вывод фигур из трех различных мест, а также подсчет фигур с площадью большей чем заданное число.

1. **Вывод**

С помощью пользовательских аллокаторов программист может более эффективно распоряжаться отданной для хранения фигур памятью, сам следить за процессом выделения и очистки памяти, конструирования и деконструирования объектов.