Московский Авиационный Институт
(Национальный исследовательский Университет)

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

Лабораторная работа №6
по курсу «ООП»

Тема:
Итераторы и аллокаторы.

Студент:	Обыденкова Ю.Ю.
Группа:	М80-208Б-18
Преподаватель:	Журавлев А.А.
Вариант:	18
Оценка:	
Дата:	

Москва
2019

1. **Код программы:**

Allocator.h

```
#pragma once

#include <memory>
#include "List.h"

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

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

template<class V>
struct rebind {
using other = Allocator<V, ALLOC_SIZE>;
};

Allocator() {
size_t object_count = ALLOC_SIZE / sizeof(T);

memory = reinterpret_cast<char*>(operator new(sizeof(T) * object_count));
for (size_t i = 0; i < object_count; ++i) {
free_blocks.Insert(free_blocks.end(), memory + sizeof(T) * i);
}
}

~Allocator() {
operator delete(memory);
}

T* allocate(size_t size) {
if (size > 1) {
throw std::logic_error("This allocator cant do that");
}
if (free_blocks.Empty()) {
throw std::bad_alloc();
}
T* temp = reinterpret_cast<T*>(*free_blocks.begin());
free_blocks.Erase(free_blocks.begin());
return temp;

}

void deallocate(T* ptr, size_t size) {
```

```
if (size > 1) {
throw std::logic_error("This allocator cant do that");
}
free_blocks.Insert(free_blocks.end(), reinterpret_cast<char*>(ptr));
}

private:
Containers::List<char*> free_blocks;
char* memory;
};
```

Point.h

```
#pragma once

#include <numeric>
#include <iostream>
#include <vector>
#include <cmath>
#include <limits>

template <typename T>
struct Point {
T x = 0;
T y = 0;
};

template <typename T>
class Vector {
public:
explicit Vector(T a, T b);
explicit Vector(Point<T> a, Point<T> b);
bool operator == (Vector rhs);
Vector operator - ();
double length() const;
T x;
T y;
};

template <typename T>
Point<T> operator + (Point<T> lhs, Point<T> rhs) {
return {lhs.x + rhs.x, lhs.y + rhs.y};
}
```

```
template <typename T>
Point<T> operator - (Point<T> lhs, Point<T> rhs) {
return {lhs.x - rhs.x, lhs.y - rhs.y};
}

template <typename T>
Point<T> operator / (Point<T> lhs, double a) {
return { lhs.x / a, lhs.y / a};
}

template <typename T>
Point<T> operator * (Point<T> lhs, double a) {
return {lhs.x * a, lhs.y * a};
}

template <typename T>
bool operator < (Point<T> lhs, Point<T> rhs) {
return (lhs.x * lhs.x + lhs.y * lhs.y) < (lhs.x * lhs.x + lhs.y * lhs.y);
}

template <typename T>
double operator * (Vector<T> lhs, Vector<T> rhs) {
return lhs.x * rhs.x + lhs.y * rhs.y;
}

template <typename T>
bool is_parallel(const Vector<T>& lhs, const Vector<T>& rhs) {
return (lhs.x * rhs.y - lhs.y * rhs.y) == 0;
}

template <typename T>
bool Vector<T>::operator == (Vector<T> rhs) {
return
std::abs(x - rhs.x) < std::numeric_limits<double>::epsilon() * 100
&& std::abs(y - rhs.y) < std::numeric_limits<double>::epsilon() * 100;
}

template <typename T>
double Vector<T>::length() const {
return sqrt(x*x + y*y);
}

template <typename T>
Vector<T>::Vector(T a, T b)
: x(a), y(b) {

}

template <typename T>
Vector<T>::Vector(Point<T> a, Point<T> b)
: x(b.x - a.x), y(b.y - a.y){
```

```
}

template <typename T>
Vector<T> Vector<T>::operator - () {
return Vector(-x, -y);
}

template <typename T>
bool is_perpendecular(const Vector<T>& lhs, const Vector<T>& rhs) {
return (lhs * rhs) == 0;
}

template <typename T>
double point_and_line_distance(Point<T> p1, Point<T> p2, Point<T> p3) {
double A = p2.y - p3.y;
double B = p3.x - p2.x;
double C = p2.x*p3.y - p3.x*p2.y;
return (std::abs(A*p1.x + B*p1.y + C) / std::sqrt(A*A + B*B));
}

template <typename T>
std::ostream& operator << (std::ostream& os, const Point<T>& p) {
return os << p.x << " " << p.y;
}

template <typename T>
std::istream& operator >> (std::istream& is, Point<T>& p) {
return is >> p.x >> p.y;
}
```

Square.h

```
#pragma once

#include <iostream>
#include <exception>
#include "Point.h"


template <typename T>
class Square {
public:
Square() = default;
Square(Point<T> p1, Point<T> p2, Point<T>p3, Point<T> p4);
Point<T> Center() const;
double Area() const;
void Print(std::ostream& os) const;
```

```
void Scan(std::istream& is);

private:
Point<T> p1_, p2_, p3_, p4_;
};

template <typename T>
Square<T>::Square(Point<T> p1, Point<T> p2, Point<T> p3, Point<T> p4)
: p1_(p1), p2_(p2), p3_(p3), p4_(p4){
Vector<T> v1(p1_, p2_), v2(p3_, p4_);
if (v1 = Vector<T>(p1_, p2_), v2 = Vector<T>(p3_, p4_), is_parallel(v1, v2)) {
if (v1 * v2 < 0) {
std::swap(p3_, p4_);
}
} else if (v1 = Vector<T>(p1_, p3_), v2 = Vector<T>(p2_, p4_), is_parallel(v1, v2)) {
if (v1 * v2 < 0) {
std::swap(p2_, p4_);
}
std::swap(p2_, p3_);
} else if (v1 = Vector<T>(p1_, p4_), v2 = Vector<T>(p2_, p3_), is_parallel(v1, v2)) {
if (v1 * v2 < 0) {
std::swap(p2_, p3_);
}
std::swap(p2_, p4_);
std::swap(p3_, p4_);
} else {
throw std::logic_error("At least 2 sides of Square must be parallel");
}

}

template <typename T>
Point<T> Square<T>::Center() const {
return (p1_ + p2_ + p3_ + p4_) / 4;
}

template<typename T>
double Square<T>::Area() const {
double height = point_and_line_distance(p1_, p3_, p4_);
return (Vector<T>(p1_, p2_).length() + Vector<T>(p3_, p4_).length()) * height / 2;
}

template<typename T>
void Square<T>::Print(std::ostream& os) const {
os << "Square p1:" << p1_ << ", p2:" << p2_ << ", p3:" << p3_ << ", p4:" << p4_;
}

template <typename T>
void Square<T>::Scan(std::istream &is) {
Point<T> p1,p2,p3,p4;
is >> p1 >> p2 >> p3 >> p4;
*this = Square(p1,p2,p3,p4);
```

```
}

template <typename T>
std::ostream& operator << (std::ostream& os, const Square<T>& trap) {
trap.Print(os);
return os;
}

template <typename T>
std::istream& operator >> (std::istream& is, Square<T>& trap) {
trap.Scan(is);
return is;
}
```

List.h

```
#pragma once

#include <memory>
#include <exception>

namespace Containers {

template <typename T>
struct ListNode {
T data;
std::shared_ptr<ListNode> next;
std::weak_ptr<ListNode> prev;
};

template <typename T>
struct ListIterator {
using value_type = T;
using reference = T&;
using pointer = T*;
using difference_type = ptrdiff_t;
using iterator_category = std::forward_iterator_tag;

ListIterator(std::shared_ptr<ListNode<T>> ptr)
: ptr_(ptr){}

T& operator * () {
std::shared_ptr<ListNode<T>> locked = ptr_.lock();
if (!locked) {
throw std::runtime_error("Iterator does not exist");
}
```

```
return locked->data;
}

T* operator -> () {
std::shared_ptr<ListNode<T>> locked = ptr_.lock();
if (!locked) {
throw std::runtime_error("Iterator does not exist");
}
return &locked->data;
}

ListIterator& operator++() {
std::shared_ptr<ListNode<T>> locked = ptr_.lock();
if (!locked || locked->next == nullptr) {
throw std::runtime_error("Out of bounds");
}
ptr_ = locked->next;
return *this;
}

const ListIterator operator++(int) {
auto copy = *this;
++(*this);
return copy;
}

bool operator == (const ListIterator& other) const {
return ptr_.lock() == other.ptr_.lock();
}

bool operator != (const ListIterator& other) const {
return !(*this == other);
}

std::weak_ptr<ListNode<T>> ptr_;
};

template <typename T, typename Allocator = std::allocator<T>>
class List {
public:
using allocator_type = typename Allocator::template rebind<ListNode<T>>::other;

struct deleter {
deleter(allocator_type* allocator) : allocator_(allocator) {}

void operator() (ListNode<T>* ptr) {
std::allocator_traits<allocator_type >::destroy(*allocator_, ptr);
allocator_->deallocate(ptr,1);
}

private:
allocator_type* allocator_;
```

```
};

List() {
ListNode<T>* ptr = allocator_.allocate(1);
std::allocator_traits<allocator_type >::construct(allocator_, ptr);
std::shared_ptr<ListNode<T>> new_elem(ptr, deleter(&allocator_));
tail = new_elem;
head = tail;
tail->next = nullptr;
}

List(const List&) = delete;
List(List&&) = delete;

bool Empty() const {
return head == tail;
}

T& operator[] (size_t index) {
if (index >= Size()) {
throw std::out_of_range("Index too big");
}
auto it = begin();
for (size_t i = 0; i < index; ++i) {
++it;
}
return *it;
}

ListIterator<T> begin() {
return ListIterator<T>(head);
}

ListIterator<T> end() {
return ListIterator<T>(tail);
}

void Insert(ListIterator<T> iter, T elem) {
ListNode<T>* ptr = allocator_.allocate(1);
std::allocator_traits<allocator_type>::construct(allocator_, ptr);
std::shared_ptr<ListNode<T>> new_elem(ptr, deleter(&allocator_));
new_elem->data = std::move(elem);
if (iter == begin()) {
new_elem->next = head;
head->prev = new_elem;
head = new_elem;
} else {
std::shared_ptr<ListNode<T>> cur_ptr = iter.ptr_.lock();
std::shared_ptr<ListNode<T>> prev_ptr = iter.ptr_.lock()->prev.lock();
prev_ptr->next = new_elem;
cur_ptr->prev = new_elem;
new_elem->next = cur_ptr;
```

```
new_elem->prev = prev_ptr;
}
}

void Erase(ListIterator<T> iter) {
if (iter == end()) {
throw std::runtime_error("Erasing end iterator");
}
std::shared_ptr<ListNode<T>> ptr = iter.ptr_.lock();
if (iter == begin()) {
head = head->next;
ptr->next = nullptr;
} else {
std::shared_ptr<ListNode<T>> prev_ptr = ptr->prev.lock();
std::shared_ptr<ListNode<T>> next_ptr = ptr->next;
prev_ptr->next = next_ptr;
next_ptr->prev = prev_ptr;
}
}

size_t Size() const {
size_t counter = 0;
ListIterator<T> begin_it(head);
ListIterator<T> end_it(tail);
while(begin_it != end_it) {
counter++;
++begin_it;
}
return counter;
}
private:
allocator_type allocator_;
std::shared_ptr<ListNode<T>> head;
std::shared_ptr<ListNode<T>> tail;
};

}
```

main.cpp

```
#include <iostream>
#include <map>
#include <string>
#include <algorithm>
#include <tuple>
#include <list>

#include "Square.h"
```

```cpp
#include "List.h"
#include "Allocator.h"

int main() {
std::string command;
Containers::List<Square<int>, Allocator<Square<int>,1000>> figures;
while (std::cin >> command) {
if (command == "add") {
size_t position;
std::cin >> position;
auto it = figures.begin();
try {
it = std::next(it, position);
} catch(std::exception& e) {
std::cout << "Position is too big\n";
continue;
}
Square<int> new_figure;
try {
std::cin >> new_figure;
figures.Insert(it, new_figure);
std::cout << new_figure << "\n";
} catch (std::exception& ex) {
std::cout << ex.what() << "\n";
}

} else if (command == "erase") {
size_t index;
std::cin >> index;
try {
auto it = std::next(figures.begin(), index);
figures.Erase(it);
} catch (...) {
std::cout << "Index is too big\n";
continue;
}
} else if (command == "size") {
std::cout << figures.Size() << "\n";
} else if (command == "print") {
std::for_each(figures.begin(), figures.end(), [] (const Square<int>& fig) {
std::cout << fig << " ";
});
std::cout << "\n";
} else if (command == "count") {
size_t required_area;
std::cin >> required_area;
std::cout << std::count_if(figures.begin(), figures.end(), [&required_area] (const
Square<int>& fig) {
return fig.Area() < required_area;
});
std::cout << "\n";
} else {
```

```
std::cout << "Incorrect command" << "\n";
std::cin.ignore(32767, '\n');
}
}
}
```

CMakeLists.txt

```
project(oop_exercise_06)

set(CMAKE_CXX_STANDARD 17)

add_executable(oop_exercise_06 main.cpp Square.h Point.h List.h Allocator.h)
```

2. **Ссылка на репозиторий:** https://github.com/obydenkova/oop_exercise_06

3. **Набор testcases:**

test_01.test

add 0 1 1 1 1 1 1 1 1

add 0 2 2 2 2 2 2 2 2

add 1 3 3 3 3 3 3 3 3

add 1 4 4 4 4 4 4 4 4

count 2

print

flex

erase 0

erase 1

print

test_01.result

Square p1:1 1, p2:1 1, p3:1 1, p4:1 1

Square p1:2 2, p2:2 2, p3:2 2, p4:2 2

Square p1:3 3, p2:3 3, p3:3 3, p4:3 3

Square p1:4 4, p2:4 4, p3:4 4, p4:4 4

0

Square p1:2 2, p2:2 2, p3:2 2, p4:2 2 Square p1:4 4, p2:4 4, p3:4 4, p4:4 4 Square p1:3 3, p2:3 3, p3:3 3, p4:3 3 Square p1:1 1, p2:1 1, p3:1 1, p4:1 1

Incorrect command

Square p1:4 4, p2:4 4, p3:4 4, p4:4 4 Square p1:1 1, p2:1 1, p3:1 1, p4:1 1

test_02.test

add 0 1 1 1 1 1 1 1 1

add 0 2 2 2 2 2 2 2 2

add 0 3 3 3 3 3 3 3 3

add 0 4 4 4 4 4 4 4 4

add 0 5 5 5 5 5 5 5 5

add 0 6 6 6 6 6 6 6 6

add 0 7 7 7 7 7 7 7 7

add 0 8 8 8 8 8 8 8 8

add 0 9 9 9 9 9 9 9 9

add 0 1 1 1 1 1 1 1 1

add 0 2 2 2 2 2 2 2 2

add 0 3 3 3 3 3 3 3 3

add 0 4 4 4 4 4 4 4 4

add 0 5 5 5 5 5 5 5 5

add 0 6 6 6 6 6 6 6 6

add 0 7 7 7 7 7 7 7 7

test_02.result

Square p1:1 1, p2:1 1, p3:1 1, p4:1 1

Square p1:2 2, p2:2 2, p3:2 2, p4:2 2

Square p1:3 3, p2:3 3, p3:3 3, p4:3 3

Square p1:4 4, p2:4 4, p3:4 4, p4:4 4

Square p1:5 5, p2:5 5, p3:5 5, p4:5 5

Square p1:6 6, p2:6 6, p3:6 6, p4:6 6

Square p1:7 7, p2:7 7, p3:7 7, p4:7 7

Square p1:8 8, p2:8 8, p3:8 8, p4:8 8

Square p1:9 9, p2:9 9, p3:9 9, p4:9 9

Square p1:1 1, p2:1 1, p3:1 1, p4:1 1

Square p1:2 2, p2:2 2, p3:2 2, p4:2 2

Square p1:3 3, p2:3 3, p3:3 3, p4:3 3

Square p1:4 4, p2:4 4, p3:4 4, p4:4 4

Square p1:5 5, p2:5 5, p3:5 5, p4:5 5

4. **Объяснение результатов работы программы:**

Все тесты завершились успешно

5. **Вывод:**

В данной лабораторной работе я освоила основы работы с аллокаторами. Аллокаторы позволяют ускорить быстродействие работы программы, кроме того позволяют корректно выделять память под тот или иной элемент, хранящийся в нашем контейнере.