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:
Основы работы с коллекциями: итераторы.

Студент:	Болдырев А. К.
Группа:	М80-206Б-18
Преподаватель:	Журавлев А.А.
Вариант:	3
Оценка:	
Дата:	

Москва
2019

1. Код программы а языке С++:

```
rec.h
#pragma once
#include "point.h"
#include "stack.h"
#include <cassert>
template <class T>
struct TRectangle {
TPoint<T> a, b, c, d;
TRectangle();
TRectangle(std::istream&);
double Square() const;
TPoint<T> Center() const;
void Print() const;
};
template <class T>
TRectangle<T>::TRectangle() {}
template <class T>
TRectangle<T>::TRectangle(std::istream& is) {
is >> a >> b >> c >> d;
TPoint<T> ab, ad, cb, cd;
ab.x = b.x - a.x;
ab.y = b.y - a.y;
ad.x = d.x - a.x;
ad.y = d.y - a.y;
cb.x = b.x - c.x;
cb.y = b.y - c.y;
cd.x = d.x - c.x;
cd.y = d.y - c.y;
if (acos((ab.x * ad.x + ab.y * ad.y) / (sqrt(ab.x * ab.x + ab.y * ab.y) * sqrt(ad.x
* ad.x + ad.y * ad.y))) / M_PI != 0.5 || acos((cb.x * cd.x + cb.y * cd.y) / (sqrt(cb.x *
cb.x + cb.y * cb.y) * sqrt(cd.x * cd.x + cd.y * cd.y))) / M_PI != 0.5) {
throw std::logic_error("it's not rectangle\n");
}
}
template <class T>
double TRectangle<T>::Square() const {
double ans = (b.x - a.x) * (c.y - a.y) - (c.x - a.x) * (b.y - a.y);return fabs(ans);
}
template <class T>
TPoint<T> TRectangle<T>::Center() const {
TPoint<T> p;
T x = (a.x + b.x + c.x + d.x) /4;
T y = (a.y + b.y + c.y + d.y) /4;
p.x = x;
```

```
p.y = y;
return p;
}
template <class T>
void TRectangle<T>::Print() const {
std::cout << a << b << c << d << "\n";
}
point.h:
#ifndef POINT_H
#define POINT_H 1
#include <iostream>
#include <algorithm>
#include <cmath>
template<class T>
struct TPoint {
TPoint() {}
TPoint(T a, T b) : x(a), y(b){}
T x;
T y;
};
template<class T>
std::ostream& operator << (std::ostream& os, const TPoint<T>& p)
{
os << p.x << " " << p.y << " ";
return os;
}template <class T>
std::istream& operator >> (std::istream& is, TPoint<T>& p)
{
is >> p.x >> p.y;
return is;
}
template <class T>
TPoint<T> operator /= ( TPoint<T>& p, int val)
{
p.x = p.x / val;
p.y = p.y / val;
return p;
}
template <class T>
TPoint<T> operator + (const TPoint<T>& p1, const TPoint<T>& p2)
{
TPoint<T> p;
p.x = p1.x + p2.x;
p.y = p1.y + p2.y;
return p;
```

```
}
template <class T>
TPoint<T> operator - (const TPoint<T> p1, const TPoint<T> p2)
{
TPoint<T> p;
p.x = p1.x - p2.x;
p.y = p1.y - p2.y;
return p;
}
#endif
stack.h
#ifndef STACK_H
#define STACK_H 1#include <memory>
#include <iostream>
#include <iterator>
namespace containers {
template <class T, class Allocator = std::allocator<T>>
class TStack {
private:
struct Node;
public:
TStack() = default;
class forward_iterator {
public:
using value_type = T;
using reference = T&;
using pointer = T*;
using difference_type = std::ptrdiff_t;
using iterator_category = std::forward_iterator_tag;
forward_iterator (Node* ptr) : ptr_(ptr) {};
T& operator* ();
forward_iterator& operator++ ();
forward_iterator operator++ (int);
bool operator== (const forward_iterator& o) const;
bool operator!= (const forward_iterator& o) const;
private:
Node* ptr_;
friend TStack;
};
forward_iterator begin();
forward_iterator end();
void pop();
T& top();
void push(const T& value);
void erase(const forward_iterator& it);
void insert(forward_iterator& it, const T& val);
void advance(forward_iterator& it, int idx);bool empty() {
return head == nullptr;
}
void print();
private:
using allocator_type = typename Allocator::template rebind<Node>::other;
struct deleter {
deleter(allocator_type* allocator) : allocator_(allocator) {}
void operator() (Node* ptr) {
if (ptr != nullptr) {
std::allocator_traits<allocator_type>::destroy(*allocator_, ptr);
allocator_->deallocate(ptr, 1);
}
}
private:
allocator_type* allocator_;
};
using unique_ptr = std::unique_ptr<Node, deleter>;
struct Node {
T value;
unique_ptr following{nullptr, deleter{nullptr}};
Node(const T& val) : value(val) {}
forward_iterator next();
};
allocator_type allocator_{};
unique_ptr head {nullptr, deleter{nullptr}};
};
template <class T, class Allocator>
typename TStack<T, Allocator>::forward_iterator TStack<T,
Allocator>::Node::next() {
return following.get();
}
template <class T, class Allocator>typename TStack<T, Allocator>::forward_iterator
TStack<T, Allocator>::begin() {
return head.get();
}
template <class T, class Allocator>
typename TStack<T, Allocator>::forward_iterator TStack<T, Allocator>::end() {
return nullptr;
}
template <class T, class Allocator>
T& TStack<T, Allocator>::forward_iterator::operator* () {
return ptr_->value;
}
```

```
template <class T, class Allocator>
typename TStack<T, Allocator>::forward_iterator& TStack<T,
Allocator>::forward_iterator::operator++ () {
*this = ptr_->next();
return *this;
}
template <class T, class Allocator>
typename TStack<T, Allocator>::forward_iterator TStack<T,
Allocator>::forward_iterator::operator++ (int) {
forward_iterator prev =*this;
++this;
return prev;
}
template <class T, class Allocator>
bool TStack<T, Allocator>::forward_iterator::operator== (const forward_iterator& o)
const{
return ptr_ == o.ptr_;
}
template <class T, class Allocator>
bool TStack<T, Allocator>::forward_iterator::operator!= (const forward_iterator& o)
const{
return ptr_ != o.ptr_;
}
template <class T, class Allocator>
void TStack<T, Allocator>::push(const T& value) {Node* NewNode = this-
>allocator_.allocate(1);
std::allocator_traits<allocator_type>::construct(this->allocator_, NewNode,
value);
auto tmp = unique_ptr(NewNode, deleter{&this->allocator_});
tmp->following = std::move(head);
head = std::move(tmp);
}
template <class T, class Allocator>
void TStack<T, Allocator>::pop() {
if (head.get() == nullptr) {
throw std::logic_error("Stack is empty\n");
} else {
head = std::move(head->following);
}
}
template <class T, class Allocator>
T& TStack<T, Allocator>::top() {
if (head.get() == nullptr) throw std::logic_error("Stack is empty\n");
return head->value;
}
```

```
template <class T, class Allocator>
void TStack<T, Allocator>::print() {
Node* tmp = head.get();
while (tmp != nullptr) {
std::cout << tmp->value << " ";
tmp = tmp->following.get();
}
}
template <class T, class Allocator>
void TStack<T, Allocator>::insert(forward_iterator& it, const T& value) {
if (it.ptr_ == head.get()) {
this->push(value);
return;
}
Node* NewNode = this-
>allocator_.allocate(1);std::allocator_traits<allocator_type>::construct(this-
>allocator_, NewNode,
value);
auto tmp = unique_ptr(NewNode, deleter{&this->allocator_});
//auto tmp = std::unique_ptr<Node>(new Node{value});
forward_iterator i = this->begin();
while (i.ptr_->following.get() != it.ptr_) {
if (i.ptr_ == nullptr && i.ptr_ != it.ptr_) throw std::logic_error("Out of
range\n");
++i;
}
if (i.ptr_->following == nullptr) {
i.ptr_->following = std::move(tmp);
return;
}
++i;
tmp->following = std::move(i.ptr_->following);
i.ptr_->following = std::move(tmp);
return;
}
template <class T, class Allocator>
void TStack<T, Allocator>::erase(const forward_iterator& it) {
if (it.ptr_ == head.get()) {
this->pop();
return;
}
auto i = this->begin();
while(i.ptr_ != nullptr && i.ptr_->next() != it.ptr_) {
++i;
}
if (i.ptr_ == nullptr) {
throw std::logic_error ("Out of range\n");
}
i.ptr_->following = std::move(it.ptr_->following);
return;
}
template <class T, class Allocator>
void TStack<T, Allocator>::advance(forward_iterator& it, int idx) {it = this->begin();
if (it.ptr_ == nullptr && idx > 0) throw std::logic_error("Out of
range(advance)\n");
int i = 0;
while (i < idx) {
if (it.ptr_->following == nullptr && i < idx - 1) {
throw std::logic_error("Out of range(advance)\n");
}
++it;
++i;
}
}
}
#endif
allocator.h:
#ifndef MY_ALLOCATOR_H
#define MY_ALLOCATOR_H 1
#include <cstdint>
#include <cstdint>
#include <exception>
#include <iostream>
#include <type_traits>
#include "queue.h"
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 U>
struct rebind {
using other = my_allocator<U, 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;
containers::TQueue<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("can`t allocate arrays");
}
if (size_t(pool_end - pool_tail) < sizeof(T)) {
if (!free_blocks.empty()) {
auto it = free_blocks.begin();
char* ptr = *it;
free_blocks.pop();
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("can`t allocate arrays, thus can`t deallocate them
too");
}
if(ptr == nullptr){
return;
}
free_blocks.push(reinterpret_cast<char*>(ptr));
}
#endif
```

main.cpp:

```
#include "stack.h"
#include "allocator.h"
#include "rec.h"
```

```
#include <algorithm>
#include <map>
#include <string>
int main() {
containers::TStack<TRectangle<int>, my_allocator<TRectangle<int>, 500>>
s;
std::string cmd;
int index;
std::cout << "push - to push figure to stack\n"
<< "insert - to insert figure to stack\n"
<< "pop - to pop figure from Stack\n"
<< "erase - to delete figure from Stack\n"
<< "top - to show first figure\n"
<< "for_each - to print figures\n"
<< "map - to show work allocator with map\n"
<< "exit - to finish execution of program\n";
while (true) {
std::cin >> cmd;
if (cmd == "push") {
std::cout << "enter coordinates\n";
TRectangle<int> fig;
try {
TRectangle<int> tmp(std::cin);
fig = tmp;
} catch(std::exception& err) {
std::cout << err.what() << std::endl;continue;
}
s.push(fig);
} else if (cmd == "insert") {
std::cout << "enter index\n";
std::cin >> index;
auto p = s.begin();
try {
s.advance(p, index);
} catch (std::exception& err) {
std::cout << err.what() << std::endl;
continue;
}
std::cout << "enter coordinates\n";
TRectangle<int> fig;
try {
TRectangle<int> tmp(std::cin);
fig = tmp;
} catch(std::exception& err) {
std::cout << err.what() << std::endl;
```

```
continue;
}
s.insert(p, fig);
} else if (cmd == "pop") {
try {
s.pop();
} catch(std::exception& err) {
std::cout << err.what() << std::endl;
continue;
}
} else if (cmd == "erase") {
std::cout << "enter index\n";
std::cin >> index;
auto p = s.begin();
try {
s.advance(p, index);
} catch (std::exception& err) {
std::cout << err.what() << std::endl;
continue;
}
try {
s.erase(p);
} catch (std::exception& err) {
std::cout << err.what() << std::endl;
continue;}
} else if (cmd == "top") {
try {
s.top();
} catch (std::exception& err) {
std::cout << err.what() << std::endl;
continue;
}
(s.top()).Print();
} else if (cmd == "for_each") {
std::for_each(s.begin(), s.end(), [] (TRectangle<int> tmp) {return
tmp.Print();});
} else if (cmd == "exit") {
break;
} else if (cmd == "map"){
std::map<int, int, std::less<>, my_allocator<std::pair<const int,
int>, 1000>> tree;
for (int i = 0; i < 6; i++) {
tree[i] = i * i;
}
std::for_each(tree.begin(), tree.end(), [](std::pair<int, int> X)
{std::cout << X.first << " " << X.second << " ";});
std::cout << std::endl;
} else {
std::cout << "Wrong comand\n";
continue;
}
}
}
```

2. Ссылка на репозитирий Github:

https://github.com/Anton-Boldyrev/oop_exercise_06

3. Набор тестов:

```
push - to push figure to stack
insert - to insert figure to stack
pop - to pop figure from Stack
erase - to delete figure from Stack
top - to show first figure
for_each - to print figures
map - to show work allocator with map
exit - to finish execution of program
push
enter coordinates
0 0 1 1 2 1 3 0
insert
enter index
1
enter coordinates
1 1 1 3 3 3 4 1
for_each
0 0 1 1 2 1 3 0
1 1 1 3 3 3 4 1
erase
enter index
1
for_each
0 0 1 1 2 1 3 0
top
0 0 1 1 2 1 3 0
pop
```

```
for_each
map
0 0 1 1 2 4 3 9 4 16 5 25
exit
```

4. Объяснение результатов работы программы:

Стек реализован в виде односвязного списка на итераторах. Аллокатор работает на этом же стеке. В main.cpp push добавляет элемент в начало стека, insert на позицию i, pop удаляет первый элемент, erase удаляет элемент по индексу i, for_each выводит координаты фигур на экран. Аллокатор совмести с std::map, что продемонстрировано при команде map.

5. Вывод

В ходе данной работы были получены навыки работы с аллокаторами. Аллокаторы позволяют ускорить быстродействие программ, а так же усилить контроль над менеджментом памяти.