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: | Суворова С. А. |
| Группа: | М80-206Б-18 |
| Преподаватель: | Журавлев А.А. |
| Вариант: | 22 |
| Оценка: |  |
| Дата: |  |

Москва

2019

1.**Код на C++:**

point.h:

#ifndef D\_POINT\_H\_

#define D\_POINT\_H\_

#include <iostream>

template<class T>

struct point {

double x,y;

point<T> point\_1(double x, double y);

};

template<class T>

point<T> point<T>::point\_1(double x, double y) {

point<T> p;

p.x=x;

p.y=y;

return p;

}

template<class T>

std::istream& operator>> (std::istream& is, point<T>& p){

is >> p.x >>p.y;

return is;

}

template<class T>

std::ostream& operator<< (std::ostream& os, const point<T>& p){

os << p.x << " " << p.y << " ";

return os;

}

template<class T>

point<T> operator+(point<T> x1,point<T> x2){

point<T> x3;

x3.x=x1.x+x2.x;

x3.y=x1.y+x2.y;

return x3;

}

template<class T>

point<T>& operator/= (point<T>& x1, int number){

x1.x=x1.x/number;

x1.y=x1.y/number;

return x1;

}

/\*

template<class T>

std::istream& operator>>(std::istream& is, point<T>& p);

template<class T>

std::ostream& operator<<(std::ostream& os,const point<T>& p);

template<class T>

point<T> operator+(point<T> x1,point<T> x2);

template<class T>

point<T>& operator/= (point<T>& x1, int number);

\*/

#endif

five\_angles.h:

#ifndef D\_FIVE\_ANGLES\_H\_

#define D\_FIVE\_ANGLES\_H\_

#include <iostream>

#include "point.h"

template<class T>

struct five\_angles {

five\_angles(std::istream &is);

point<T> center() const ;

void print() const ;

double square() const ;

point<T> one,two,three,four,five;

};

template<class T>

five\_angles<T>::five\_angles(std::istream &is){

is >> one >> two >> three >> four >> five;

}

template<class T>

point<T> five\_angles<T>::center() const {

point<T> p;

p=one+two+three+four+five;

p/=5;

return p;

}

template<class T>

void five\_angles<T>::print() const {

std::cout << one << " " << two << " " << three << " " << four << " " << five <<"\n";

}

template<class T>

double five\_angles<T>::square() const {

double s=0;

s=(one.x\*two.y+two.x\*three.y+three.x\*four.y+four.x\*five.y+five.x\*one.y-two.x\*one.y-

three.x\*two.y-four.x\*three.y-five.x\*four.y-one.x\*five.y)/2;

if(s<0){

return -s;

}else {

return s;

}

}

#endif

list.h:

#ifndef D\_LIST\_H\_

#define D\_LIST\_H\_

#include <iostream>

#include "five\_angles.h"

#include <memory>

#include <functional>

#include <cassert>

#include <iterator>

namespace containersl {

template<class T, class Allocator = std::allocator<T>>

struct list {

private:

struct node;

public:

list() = default;

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;

forward\_iterator(node \*ptr);

T &operator\*();

forward\_iterator &operator++();

forward\_iterator operator+(int r);

bool operator==(const forward\_iterator &o) const;

bool operator!=(const forward\_iterator &o) const;

private:

node \*ptr\_;

friend list;

};

forward\_iterator begin();

forward\_iterator end();

void insert(const forward\_iterator &it, const T &value);

void erase(const forward\_iterator &it);

int is\_empty(){

return root==nullptr;

};

size\_t size=0;

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>;

node \*end\_node = nullptr;

node \*end\_help(node \*ptr);

struct node {

T value;

unique\_ptr next{nullptr, deleter{nullptr}};

node \*parent = nullptr;

forward\_iterator nextf();

};

allocator\_type allocator\_{};

unique\_ptr root{nullptr, deleter{nullptr}};

};

//

template<class T,class Allocator>

typename list<T,Allocator>::node \*list<T,Allocator>::end\_help(containersl::list<T,Allocator>::node \*ptr) {

if ((ptr == nullptr) || (ptr->next == nullptr)) {

return ptr;

}

return list<T,Allocator>::end\_help(ptr->next.get());

}

template<class T,class Allocator>

typename list<T,Allocator>::forward\_iterator list<T,Allocator>::begin() {

if (root == nullptr) {

return nullptr;

}

forward\_iterator it(root.get());

return it;

}

template<class T,class Allocator>

typename list<T,Allocator>::forward\_iterator list<T,Allocator>::end() {

return nullptr;

}

template<class T,class Allocator>

void list<T,Allocator>::insert(const list<T,Allocator>::forward\_iterator &it, const T &value) {

std::unique\_ptr<node,list<T, Allocator>::deleter> new\_node{new node{value},deleter{&allocator\_}};

if (it != nullptr) {

node \*ptr = it.ptr\_->parent;

new\_node->parent = it.ptr\_->parent;

it.ptr\_->parent = new\_node.get();

if (ptr) {

new\_node->next = std::move(ptr->next);

ptr->next = std::move(new\_node);

} else {

new\_node->next = std::move(root);

root = std::move(new\_node);

}

} else {

new\_node->next = nullptr;

if(end\_node==nullptr) {

new\_node->parent= nullptr;

new\_node->next= nullptr;

list<T,Allocator>::root = std::move(new\_node);

}else{

new\_node->parent=end\_node;

new\_node->next= nullptr;

end\_node->next=std::move(new\_node);

}

}

end\_node = end\_help(root.get());

++size;

}

template<class T,class Allocator>

void list<T,Allocator>::erase(const list<T,Allocator>::forward\_iterator &it) {

if (it.ptr\_ == nullptr) {

throw std::logic\_error("erasing invalid iterator");

}

unique\_ptr &pointer\_from\_parent = [&]() -> unique\_ptr & {

if (it.ptr\_ == root.get()) {

return root;

}

return it.ptr\_->parent->next;

}();

pointer\_from\_parent = std::move(it.ptr\_->next);

end\_node = end\_help(root.get());

--size;

}

//

template<class T,class Allocator>

typename list<T,Allocator>::forward\_iterator list<T,Allocator>::node::nextf() {

forward\_iterator result(this->next.get());

return result;

}

template<class T,class Allocator>

list<T,Allocator>::forward\_iterator::forward\_iterator(node \*ptr): ptr\_{ptr} {}

template<class T,class Allocator>

T &list<T,Allocator>::forward\_iterator::operator\*() {

return ptr\_->value;

}

template<class T,class Allocator>

typename list<T,Allocator>::forward\_iterator &list<T,Allocator>::forward\_iterator::operator++() {

if (\*this != nullptr) {

\*this = ptr\_->nextf();

return \*this;

} else {

throw std::logic\_error("invalid iterator");

}

}

template<class T,class Allocator>

typename list<T,Allocator>::forward\_iterator list<T,Allocator>::forward\_iterator::operator+(int r) {

for (int i = 0; i < r; ++i) {

++\*this;

}

return \*this;

}

template<class T,class Allocator>

bool list<T,Allocator>::forward\_iterator::operator==(const forward\_iterator &o) const {

return ptr\_ == o.ptr\_;

}

template<class T,class Allocator>

bool list<T,Allocator>::forward\_iterator::operator!=(const forward\_iterator &o) const {

return ptr\_ != o.ptr\_;

}

}

#endif

allocater.h:

#ifndef D\_ALLOCATOR\_H\_

#define D\_ALLOCATOR\_H\_

#include <iostream>

#include "queue.h"

#include "list.h"

template<class T, size\_t ALLOC\_SIZE>

struct q\_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 = q\_allocator<U, ALLOC\_SIZE>;

};

q\_allocator() :

memory\_pool\_begin\_(new char[ALLOC\_SIZE]),

memory\_pool\_end\_(memory\_pool\_begin\_ + ALLOC\_SIZE),

memory\_pool\_tail\_(memory\_pool\_begin\_) {}

q\_allocator(const q\_allocator &) = delete;

q\_allocator(q\_allocator &&) = delete;

~q\_allocator() {

delete[] memory\_pool\_begin\_;

}

T \*allocate(std::size\_t n);

void deallocate(T \*ptr, std::size\_t n);

private:

char \*memory\_pool\_begin\_;

char \*memory\_pool\_end\_;

char \*memory\_pool\_tail\_;

containersl::list<char \*> free\_blocks\_;

};

template<class T, size\_t ALLOC\_SIZE>

T\* q\_allocator<T, ALLOC\_SIZE>::allocate(std::size\_t n) {

if(size\_t(memory\_pool\_end\_ - memory\_pool\_tail\_) < sizeof(T)){

if(!free\_blocks\_.is\_empty()){

char \*ptr;

for (size\_t i = 0; i < n; ++i) {

auto it = free\_blocks\_.begin();

ptr = \*it;

free\_blocks\_.erase(it);

}

return reinterpret\_cast<T \*>(ptr);

}

throw std::bad\_alloc();

}

T\* result = reinterpret\_cast<T\*>(memory\_pool\_tail\_);

memory\_pool\_tail\_ += sizeof(T);

return result;

}

template<class T, size\_t ALLOC\_SIZE>

void q\_allocator<T, ALLOC\_SIZE>::deallocate(T\* ptr, std::size\_t n) {

if(ptr == nullptr){

return;

}

free\_blocks\_.insert(free\_blocks\_.end(),reinterpret\_cast<char\*>(ptr));

memory\_pool\_tail\_ -= sizeof(T);

}

#endif

queue.h:

#ifndef D\_QUEUE\_H\_

#define D\_QUEUE\_H\_

#include <iostream>

#include "five\_angles.h"

#include <memory>

#include <functional>

#include <cassert>

#include <iterator>

#include <type\_traits>

namespace containers {

template<class T,class Allocator>

struct queue {

private:

struct node;

public:

queue() = default;

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;

forward\_iterator(node \*ptr);

T &operator\*();

forward\_iterator &operator++();

forward\_iterator operator+(int r);

bool operator==(const forward\_iterator &o) const;

bool operator!=(const forward\_iterator &o) const;

private:

node \*ptr\_;

friend queue;

};

forward\_iterator begin();

forward\_iterator end();

void insert(const forward\_iterator &it, const T &value);

void erase(const forward\_iterator &it);

void pop();

void push(const T &value);

T front();

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>;

node \*end\_node = nullptr;

node \*end\_help(node \*ptr);

struct node {

T value;

unique\_ptr next{nullptr, deleter{nullptr}};

node \*parent = nullptr;

forward\_iterator nextf();

};

allocator\_type allocator\_{};

unique\_ptr root{nullptr, deleter{nullptr}};

};

//

template<class T,class Allocater>

typename queue<T,Allocater>::node \*queue<T,Allocater>::end\_help(containers::queue<T,Allocater>::node \*ptr) {

if ((ptr == nullptr) || (ptr->next == nullptr)) {

return ptr;

}

return queue<T,Allocater>::end\_help(ptr->next.get());

}

template<class T,class Allocater>

typename queue<T,Allocater>::forward\_iterator queue<T,Allocater>::begin() {

if (root == nullptr) {

return nullptr;

}

forward\_iterator it(root.get());

return it;

}

template<class T,class Allocater>

typename queue<T,Allocater>::forward\_iterator queue<T,Allocater>::end() {

return nullptr;

}

template<class T,class Allocator>

void queue<T,Allocator>::insert(const queue<T,Allocator>::forward\_iterator &it, const T &value) {

node\* ptr\_result =allocator\_.allocate(1);

ptr\_result->value=value;

//std::allocator\_traits<allocator\_type>::construct(allocator\_,ptr\_result, value);

std::unique\_ptr<node,queue<T, Allocator>::deleter> new\_node(ptr\_result,deleter{&allocator\_});

if (it != nullptr) {

node \*ptr = it.ptr\_->parent;

new\_node->parent = it.ptr\_->parent;

it.ptr\_->parent = new\_node.get();

if (ptr) {

new\_node->next = std::move(ptr->next);

ptr->next = std::move(new\_node);

} else {

new\_node->next = std::move(root);

root = std::move(new\_node);

}

} else {

new\_node->next = nullptr;

if(end\_node==nullptr) {

new\_node->parent= nullptr;

new\_node->next= nullptr;

queue<T,Allocator>::root = std::move(new\_node);

}else{

new\_node->parent=end\_node;

new\_node->next= nullptr;

end\_node->next=std::move(new\_node);

}

}

end\_node = end\_help(root.get());

}

template<class T,class Allocater>

void queue<T,Allocater>::erase(const queue<T,Allocater>::forward\_iterator &it) {

if (it.ptr\_ == nullptr) {

throw std::logic\_error("erasing invalid iterator");

}

unique\_ptr &pointer\_from\_parent = [&]() -> unique\_ptr & {

if (it.ptr\_ == root.get()) {

return root;

}

return it.ptr\_->parent->next;

}();

if(it.ptr\_->next) {

it.ptr\_->next->parent = it.ptr\_->parent;

}

pointer\_from\_parent = std::move(it.ptr\_->next);

end\_node = end\_help(root.get());

}

//

template<class T,class Allocater>

typename queue<T,Allocater>::forward\_iterator queue<T,Allocater>::node::nextf() {

forward\_iterator result(this->next.get());

return result;

}

template<class T,class Allocater>

queue<T,Allocater>::forward\_iterator::forward\_iterator(node \*ptr): ptr\_{ptr} {}

template<class T,class Allocater>

T &queue<T,Allocater>::forward\_iterator::operator\*() {

return ptr\_->value;

}

template<class T,class Allocater>

typename queue<T,Allocater>::forward\_iterator &queue<T,Allocater>::forward\_iterator::operator++() {

if (\*this != nullptr) {

\*this = ptr\_->nextf();

return \*this;

} else {

throw std::logic\_error("invalid iterator");

}

}

template<class T,class Allocater>

typename queue<T,Allocater>::forward\_iterator queue<T,Allocater>::forward\_iterator::operator+(int r) {

for (int i = 0; i < r; ++i) {

++\*this;

}

return \*this;

}

template<class T,class Allocater>

bool queue<T,Allocater>::forward\_iterator::operator==(const forward\_iterator &o) const {

return ptr\_ == o.ptr\_;

}

template<class T,class Allocater>

bool queue<T,Allocater>::forward\_iterator::operator!=(const forward\_iterator &o) const {

return ptr\_ != o.ptr\_;

}

template<class T,class Allocater>

T queue<T,Allocater>::front() {

if (queue<T,Allocater>::root == nullptr) {

throw std::logic\_error("no elements");

}

return queue<T,Allocater>::root->value;

}

template<class T,class Allocater>

void queue<T,Allocater>::pop() {

if (queue<T,Allocater>::root == nullptr) {

throw std::logic\_error("no elements");

}

erase(queue<T,Allocater>::begin());

}

template<class T,class Allocater>

void queue<T,Allocater>::push(const T &value) {

forward\_iterator it(end\_node);

node\* ptr\_result =allocator\_.allocate(1);

ptr\_result->value=value;

//std::allocator\_traits<allocator\_type>::construct(allocator\_,ptr\_result, value);

std::unique\_ptr<node,queue<T, Allocater>::deleter> new\_node(ptr\_result,deleter{&allocator\_});

if (it.ptr\_) {

new\_node->parent = it.ptr\_;

it.ptr\_->next = std::move(new\_node);

} else {

new\_node->next = nullptr;

queue<T,Allocater>::root = std::move(new\_node);

}

queue<T,Allocater>::end\_node = end\_help(root.get());

}

}

#endif

main.cpp:

#include <iostream>

#include "five\_angles.h"

#include "point.h"

#include "queue.h"

#include <string.h>

#include <algorithm>

#include "allocator.h"

#include <map>

int main() {

char str[10];

containers::queue<five\_angles<double>,q\_allocator<five\_angles<double >,5\*sizeof(five\_angles<double>)>> q;

while(std::cin >> str){

if(strcmp(str,"push")==0){

five\_angles<double> five\_angle(std::cin);

try {

q.push(five\_angle);

}catch (std::exception& ex){

std::cout <<ex.what() << "\n";

}

}else if(strcmp(str,"pop")==0){

try {

q.pop();

std::cout << "\n";

}catch (std::exception& ex){

std::cout <<ex.what() << "\n";

}

}else if(strcmp(str,"front")==0){

try {

q.front().print();

std::cout << "\n";

}catch (std::exception& ex){

std::cout <<ex.what() << "\n";

}

}else if(strcmp(str,"square")==0){

int g;

std::cin >> g;

long res=std::count\_if(q.begin(),q.end(),[g](five\_angles<double> f){ return f.square() < g;});

std::cout << res << "\n";

}else if(strcmp(str,"erase")==0){

int r;

std::cin >>r;

try {

q.erase(q.begin() + r);

}catch(std::exception& ex){

std::cout <<ex.what() << "\n";

}

}else if(strcmp(str,"insert")==0){

int r;

std::cin >>r;

five\_angles<double> five\_angle(std::cin);

try {

q.insert(q.begin() + r, five\_angle);

}catch (std::exception& ex){

std::cout <<ex.what() << "\n";

}

}else if(strcmp(str,"all")==0){

std::for\_each(q.begin(),q.end(),[](five\_angles<double> f){f.print(); });

std::cout<< "\n";

}

}

return 0;

}

2. **Ссылка на репозиторий в GitHub:**

https://github.com/Suvorova-Sofya/oop\_exercise\_06

3.**Набор testcases:**

test1:

pop  
no elements  
push 1 1 2 2 3 3 4 4 5 5  
pop  
1 1 2 2 3 3 4 4 5 5   
pop  
no elements

test2:

push 1 1 2 2 3 3 4 4 5 5  
push 2 2 3 3 4 4 5 5 6 6  
all  
1 1 2 2 3 3 4 4 5 5   
2 2 3 3 4 4 5 5 6 6

test3:

push 1 1 2 2 3 3 4 4 5 5  
push 2 2 3 3 4 4 5 5 6 6  
square 10  
2  
square 0  
0

4.**Результаты выполнения программы:**

test1:

pop  
no elements  
push 1 1 2 2 3 3 4 4 5 5  
pop  
1 1 2 2 3 3 4 4 5 5 

pop  
no elements

test2:

push 1 1 2 2 3 3 4 4 5 5  
push 2 2 3 3 4 4 5 5 6 6  
all  
1 1 2 2 3 3 4 4 5 5   
2 2 3 3 4 4 5 5 6 6

test3:

push 1 1 2 2 3 3 4 4 5 5  
push 2 2 3 3 4 4 5 5 6 6  
square 10  
2  
square 0  
0

5**. Объяснение результатов работы программы:**

Пользователь вводит команду , и если команда была push -координаты фигуры. Далее программа выполняет определенное действие с очередью взависимости от команды и либо возвращает определенное значение ,либо нет. Также пользователь должен знать какой объем памяти дал ему аллокатор, чтобы не выйти за её пределы.

6.**Вывод:**

В данной программе показывается ,каким образом можно создать собственный аллокатор, чтобы лучше понимать каким образом работает данная структура данных, и следовательно знать наиболее эффективный способ её использования.