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 С++:**

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.