Московский Авиационный Институт
(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Лабораторная работа
по курсу «Объектно-ориентированное программирование»
III Семестр

Задание 6
Вариант 24
Основы работы с коллекциями: аллокаторы

Студент:	Синявский А.В
Группа:	М80-208Б-18
Преподаватель:	Журавлёв А.А
Оценка:	
Дата:	

Москва 2019

1. Код программы на языке C++
1.1 vertex.h

```
#ifndef OOP_LAB5_VERTEX_H
#define OOP_LAB5_VERTEX_H

#include <iostream>
#include <type_traits>
#include <cmath>


template<class T>
struct vertex {
    T x;
    T y;
    vertex<T>& operator=(vertex<T> A);
};

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

template<class T>
std::ostream& operator<<(std::ostream& os, vertex<T> p) {
    os << '(' << p.x << ' ' << p.y << ')';
    return os;
}

template<class T>
vertex<T> operator+(const vertex<T>& A, const vertex<T>& B) {
    vertex<T> res;
    res.x = A.x + B.x;
    res.y = A.y + B.y;
    return res;
}

template<class T>
vertex<T>& vertex<T>::operator=(const vertex<T> A) {
    this->x = A.x;
    this->y = A.y;
    return *this;
}

template<class T>
vertex<T> operator+=(vertex<T> &A, const vertex<T> &B) {
    A.x += B.x;
    A.y += B.y;
    return A;
}
```

```
template<class T>
vertex<T> operator/=(vertex<T>& A, const double B) {
   A.x /= B;
   A.y /= B;
}

template<class T>
double vert_length(vertex<T>& A, vertex<T>& B) {
   double res = sqrt( pow(B.x - A.x, 2) + pow(B.y - A.y, 2) );
   return res;
}

template<class T>
struct is_vertex : std::false_type {};

template<class T>
struct is_vertex<vertex<T>> : std::true_type {};

#endif //OOP_LAB5_VERTEX_H
```

1.2 octagon.h

```
#ifndef OOP_LAB5_OCTAGON_H
#define OOP_LAB5_OCTAGON_H

#include "vertex.h"

template <class T>
class Octagon {
public:
   vertex<T> dots[8];

   explicit Octagon<T>(std::istream& is) {
      for (auto & dot : dots) {
         is >> dot;
      }
   }

   Octagon<T>() = default;

   double Area() {
      double res = 0;
      for (size_t i = 0; i < 7; i++) {
         res += (dots[i].x * dots[i+1].y) - (dots[i+1].x * dots[i].y);
      }
      res = res + (dots[7].x * dots[0].y) - (dots[0].x * dots[7].y);
      return std::abs(res)/ 2;
   }

   void Printout(std::ostream& os) {
      for (int i = 0; i < 8; ++i) {
         os << this->dots[i];
         if (i != 7) {
```

```
            os << ", ";
        }
    }
    os << std::endl;
}

void operator<< (std::ostream& os) {
    for (int i = 0; i < 8; ++i) {
        os << this->dots[i];
        if (i != 7) {
            os << ", ";
        }
    }
}
};

#endif //OOP_LAB5_OCTAGON_H
```

1.3 queue.h

```
#ifndef OOP_EXERCISE_05_QUEUE_H
#define OOP_EXERCISE_05_QUEUE_H

#include <iterator>
#include <memory>

namespace containers {
    //namespace
    //
    template<class T, class Allocator = std::allocator<T>>
    class queue {
    private:
        struct element;
        size_t size = 0;
    public:
        queue() = 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;
            explicit forward_iterator(element* ptr);
            T& operator*();
            forward_iterator& operator++();
            forward_iterator operator++(int);
            bool operator== (const forward_iterator& other) const;
            bool operator!= (const forward_iterator& other) const;
```

```
        private:
            element* it_ptr;
            friend queue;
        };

        forward_iterator begin();
        forward_iterator end();
        void push(const T& value);
        T& top();
        void pop();
        size_t length();
        void delete_by_it(forward_iterator d_it);
        void delete_by_number(size_t N);
        void insert_by_it(forward_iterator ins_it, T& value);
        void insert_by_number(size_t N, T& value);
    private:
        using allocator_type = typename Allocator::template rebind<element>::other;

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

            void operator() (element* 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<element, deleter>;

        struct element {
            T value;
            unique_ptr next_element{nullptr, deleter{nullptr}};
            element(const T& value_): value(value_) {}
            forward_iterator next();
        };

        allocator_type allocator_{};
        unique_ptr first{nullptr, deleter{nullptr}};
        element* tail = nullptr;
    };//==============================end-of-class-
queue=================================//

    template<class T, class Allocator>
    typename queue<T, Allocator>::forward_iterator queue<T, Allocator>::begin() {
        return forward_iterator(first.get());
    }
```

```
    template<class T, class Allocator>
    typename queue<T, Allocator>::forward_iterator queue<T, Allocator>::end() {
        return forward_iterator(nullptr);
    }
//=========================base-methods-of-
queue=========================================//
    template<class T, class Allocator>
    size_t queue<T, Allocator>::length() {
      return size;
    }

    template<class T, class Allocator>
    void queue<T, Allocator>::push(const T &value) {
        element* result = this->allocator_.allocate(1);
        std::allocator_traits<allocator_type>::construct(this->allocator_, result, value);
        if (!size) {
            first = unique_ptr(result, deleter{&this->allocator_});
            tail = first.get();
            size++;
            return;
        }
        tail->next_element = unique_ptr(result, deleter{&this->allocator_});
        tail = tail->next_element.get();
        size++;
    }

    template<class T, class Allocator>
    void queue<T, Allocator>::pop() {
        if (size == 0) {
            throw std::logic_error ("can`t pop from empty queue");
        }
        first = std::move(first->next_element);
        size--;
    }

    template<class T, class Allocator>
    T& queue<T, Allocator>::top() {
        if (size == 0) {
            throw std::logic_error ("queue is empty, lol, it has no top");
        }
        return first->value;
    }
//================================advanced-
methods=======================================//

    template<class T, class Allocator>
    void queue<T, Allocator>::delete_by_it(containers::queue<T, Allocator>::forward_iterator d_it) {
        forward_iterator i = this->begin(), end = this->end();
        if (d_it == end) throw std::logic_error ("out of borders");
        if (d_it == this->begin()) {
            this->pop();
            return;
```

```
        }
        while((i.it_ptr != nullptr) && (i.it_ptr->next() != d_it)) {
          ++i;
        }
        if (i.it_ptr == nullptr) throw std::logic_error ("out of borders");
        i.it_ptr->next_element = std::move(d_it.it_ptr->next_element);
        size--;
    }

    template<class T, class Allocator>
    void queue<T, Allocator>::delete_by_number(size_t N) {
        N++;
        forward_iterator it = this->begin();
        for (size_t i = 1; i <= N; ++i) {
            if (i == N) break;
            ++it;
        }
        this->delete_by_it(it);
    }

    template<class T, class Allocator>
    void queue<T, Allocator>::insert_by_it(containers::queue<T, Allocator>::forward_iterator ins_it,
T& value) {
        auto tmp = std::unique_ptr<element>(new element{value});
        forward_iterator i = this->begin();
        if (ins_it == this->begin()) {
            tmp->next_element = std::move(first);
            first = std::move(tmp);
            size++;
            return;
        }
        while((i.it_ptr != nullptr) && (i.it_ptr->next() != ins_it)) {
            ++i;
        }
        if (i.it_ptr == nullptr) throw std::logic_error ("out of borders");
        tmp->next_element = std::move(i.it_ptr->next_element);
        i.it_ptr->next_element = std::move(tmp);
        size++;
    }

    template<class T, class Allocator>
    void queue<T, Allocator>::insert_by_number(size_t N, T& value) {
        forward_iterator it = this->begin();
        for (size_t i = 1; i <= N; ++i) {
            if (i == N) break;
            ++it;
        }
        this->insert_by_it(it, value);
    }
//============================iterator`s-
stuff=====================================//
    template<class T, class Allocator>
```

```
    typename queue<T, Allocator>::forward_iterator queue<T, Allocator>::element::next() {
        return forward_iterator(this->next_element.get());
    }

    template<class T, class Allocator>
    queue<T, Allocator>::forward_iterator::forward_iterator(containers::queue<T,
Allocator>::element *ptr) {
        it_ptr = ptr;
    }

    template<class T, class Allocator>
    T& queue<T, Allocator>::forward_iterator::operator*() {
        return this->it_ptr->value;
    }

    template<class T, class Allocator>
    typename queue<T, Allocator>::forward_iterator& queue<T,
Allocator>::forward_iterator::operator++() {
        if (it_ptr == nullptr) throw std::logic_error ("out of queue borders");
        *this = it_ptr->next();
        return *this;
    }

    template<class T, class Allocator>
    typename queue<T, Allocator>::forward_iterator queue<T,
Allocator>::forward_iterator::operator++(int) {
        forward_iterator old = *this;
        ++*this;
        return old;
    }

    template<class T, class Allocator>
    bool queue<T, Allocator>::forward_iterator::operator==(const forward_iterator& other) const {
        return it_ptr == other.it_ptr;
    }

    template<class T, class Allocator>
    bool queue<T, Allocator>::forward_iterator::operator!=(const forward_iterator& other) const {
        return it_ptr != other.it_ptr;
    }

    //
    //namespace
}



#endif //OOP_EXERCISE_05_QUEUE_H
```

1.4 allocator.h

```
#ifndef D_ALLOCATOR_H_
#define D_ALLOCATOR_H_

#include <cstdlib>
#include <iostream>
#include <type_traits>
#include "../containers/queue.h"

namespace allocators {
    /*
     * NAMESPACE
     */

    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::queue<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.length()) {
                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));
    }

    /*
     * NAMESPACE
     */
}

#endif // D_ALLOCATOR_H_
```

1.5 main.cpp

```
#include <iostream>
#include <algorithm>
#include <map>

#include "octagon.h"
#include "containers/queue.h"
#include "allocators/allocator.h"

int main() {
    size_t N;
    char option = '0';
    containers::queue<Octagon<int>, allocators::my_allocator<Octagon<int>, 800>> q;
    Octagon<int> oct{};
    while (option != 'q') {
```

```
        std::cout << "choose option (m for man, q to quit)" << std::endl;
        std:: cin >> option;
        switch (option) {
            case 'q':
                break;
            case 'm':
                std::cout << "1) push new element into queue\n"
                          << "2) pop element from the queue\n"
                          << "3) delete element from the chosen position\n"
                          << "4) print queue\n"
                          << "5) option fore code testing (for example compatibility with std::map)\n"
                          << std::endl;
                break;
            case '1': {
                std::cout << "enter octagon (have to enter dots consequently): " << std::endl;
                oct = Octagon<int>(std::cin);
                q.push(oct);
                break;
            }
            case '2': {
                q.pop();
                break;
            }
            case '3': {
                std::cout << "enter position to delete: ";
                std::cin >> N;
                q.delete_by_number(N);
                break;
            }
            case '4': {
                std::for_each(q.begin(), q.end(), [](Octagon<int> &X) { X.Printout(std::cout); });
                break;
            }
            case '5': {
                std::map<int, int, std::less<>, allocators::my_allocator<std::pair<const int, int>, 100>>
mp;
                for(int i = 0; i < 2; ++i){
                    mp[i] = i * i;
                }
                std::for_each(mp.begin(), mp.end(), [](std::pair<int, int> X) { std::cout << X.first << ' '
<< X.second << ",  "; });
                std::cout << std::endl;
                for(int i = 2; i < 10; ++i){
                    mp.erase(i - 2);
                    mp[i] = i * i;
                }
                std::for_each(mp.begin(), mp.end(), [](std::pair<int, int> X) { std::cout << X.first << ' '
<< X.second << ",  "; });
                std::cout << std::endl;
                break;
            }
            default:
```

```
            std::cout << "no such option. Try m for man" << std::endl;
            break;
        }
    }
    return 0;
}
```

2. Ссылка на репозиторий на GitHub

https://github.com/Siegmeyer1/oop_exercise_06

3. Набор тестов

```
1)
m
1
2 0   4 0   6 1   6 3   4 4   2 4   0 3   0 1
1
0 0   2 0   4 0   4 2   4 4   2 4   0 4   0 2
1
0 0   1 0   2 0   2 1   2 2   1 2   0 2   0 1
4
3
2
4
2
4
q


2)
5
q
```

4. Результаты тестов

1)

```
choose option (m for man, q to quit)
m
1) push new element into queue
2) pop element from the queue
3) delete element from the chosen position
4) print queue
5) option fore code testing (for example compatibility with std::map)
```

```
choose option (m for man, q to quit)
1
enter octagon (have to enter dots consequently):
2 0    4 0    6 1    6 3    4 4    2 4    0 3    0 1
choose option (m for man, q to quit)
1
enter octagon (have to enter dots consequently):
0 0    2 0    4 0    4 2    4 4    2 4    0 4    0 2
choose option (m for man, q to quit)
1
enter octagon (have to enter dots consequently):
0 0    1 0    2 0    2 1    2 2    1 2    0 2    0 1
choose option (m for man, q to quit)
4
(2 0), (4 0), (6 1), (6 3), (4 4), (2 4), (0 3), (0 1)
(0 0), (2 0), (4 0), (4 2), (4 4), (2 4), (0 4), (0 2)
(0 0), (1 0), (2 0), (2 1), (2 2), (1 2), (0 2), (0 1)
choose option (m for man, q to quit)
3
enter position to delete: 2
choose option (m for man, q to quit)
4
(2 0), (4 0), (6 1), (6 3), (4 4), (2 4), (0 3), (0 1)
(0 0), (1 0), (2 0), (2 1), (2 2), (1 2), (0 2), (0 1)
choose option (m for man, q to quit)
2
choose option (m for man, q to quit)
4
(0 0), (1 0), (2 0), (2 1), (2 2), (1 2), (0 2), (0 1)
choose option (m for man, q to quit)
q
anri@andrew-HP-250-G6:~/Documents/Github_repositories/OOP_lab6/build$
```

2)

```
choose option (m for man, q to quit)
5
0 0,  1 1,
8 64,  9 81,
choose option (m for man, q to quit)
q
anri@andrew-HP-250-G6:~/Documents/Github_repositories/OOP_lab6/build$
```

5. Объяснение работы программы

Аллокатор используется очередью из предыдущей лабораторной для выделения памяти. Аллокатор удовлетворяет требованиям к аллокаторам, а значит совместим с std::map, что демонстрируется на простом примере в блоке

main. Метод deallocate вызывается автоматически, так как находится внутри деструктора, используемого unique_ptr-ом внутри самой очереди.

Вывод

Проделав работу, я узнал, что такое аллокаторы памяти, зачем они нужны и какие они бывают, а также реализовал собственный аллокатор. Узнал, как сделать собственный аллокатор совместимым с коллекциями стандартной библиотеки шаблонов.