Московский Авиационный Институт
(Национальный Исследовательский Университет)

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

Лабораторная работа
Дисциплина: «Объектно-ориентированное программирование»
III семестр
Задание 6: «Основы работы с коллекциями: итераторы»

Группа:	М8О-208Б-18, №3
Студент:	Анисимов Валерий Алексеевич
Преподаватель:	Журавлёв Андрей Андреевич
Оценка:	
Дата:	25.11.2019

Москва, 2019

1. **Тема**: Основы работы с коллекциями: итераторы

2. **Цель работы**: Изучение основ работы с контейнерами, знакомство с концепцией аллокаторов памяти

3. **Задание** (*вариант № 3*):
Фигура — прямоугольник. Контейнер — стек. Аллокатор — стек.

4. **Адрес репозитория на GitHub**
   https://github.com/wAlienUFOx/oop_exercise_06

5. **Код программы на С++**

main.cpp

```
#include <iostream>
#include <algorithm>
#include <map>
#include "rectangle.h"
#include "containers/stack.h"
#include "allocators/allocator.h"

int main() {
   size_t N;
   float S;
   char option = '0';
   containers::stack<Rectangle<int>, allocators::my_allocator<Rectangle<int>,
800>> st;
   Rectangle<int> *rec = nullptr;
   while (option != 'q') {
      std::cout << "choose option (m - man)" << std::endl;
      std::cin >> option;
      switch (option) {
         case 'q':
            break;
         case 'm':
            std::cout << "1) push new element into stack\n"
                      << "2) insert element into chosen position\n"
                      << "3) pop element from the stack\n"
                      << "4) delete element from the chosen position\n"
                      << "5) print stack\n"
                      << "6) count elements with area less then chosen value\n"
                      << "7) print top element\n"
                      << "8) test containers\n"
                      << "q) - quit" << std::endl;
            break;
        case '1': {
            try{
                rec =  new Rectangle<int>(std::cin);
            }catch(std::logic_error& err){
                std::cout << err.what() << std::endl;
                break;
            }
            st.push(*rec);
            break;
        }
        case '2': {
            std::cout << "enter position to insert" << std::endl;
            std::cin >> N;
            std::cout << "enter rectangle" << std::endl;
            try{
                rec =  new Rectangle<int>(std::cin);
            }catch(std::logic_error& err){
                std::cout << err.what() << std::endl;
                break;
            }
            st.insert_by_number(N, *rec);
            break;
        }
        case '3': {
            st.pop();
            break;
        }
        case '4': {
            std::cout << "enter position to delete" << std::endl;
            std::cin >> N;
            st.delete_by_number(N);
            break;
        }
        case '5': {
            std::for_each(st.begin(), st.end(), [](Rectangle<int> &REC)
{ REC.Print(std::cout); });
            break;
        }
        case '6': {
            std::cout << "enter max area" <<std::endl;
            std::cin >> S;
            std::cout << std::count_if(st.begin(), st.end(), [=](Rectangle<int> &X)
{ return X.Area() < S; })
                        << std::endl;
            break;
        }
        case '7' : {
            st.top().Print(std::cout);
            break;
        }
        case '8': {
            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 << "Wrong. Try m - manual" << std::endl;
            break;
    }
}
return 0;
}
```

point.h

```
#ifndef OOP_LAB5_POINT_H
#define OOP_LAB5_POINT_H

#include <iostream


template<class T>
struct point {
    T x;
    T y;
};

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, point<T> p) {
    os << '(' << p.x << ' ' << p.y << ')';
    return os;
}

#endif
```

rectangle.cpp

```
#ifndef OOP_LAB5_RECTANGLE_H
#define OOP_LAB5_RECTANGLE_H

#include "point.h"
#include <cmath>

template <class T>
class Rectangle {
public:
    point<T> A , B, C, D;

    explicit Rectangle<T>(std::istream& is) {
        is >> A >> B >> C >> D;
        double a, b, c, d, d1, d2, ABC, BCD, CDA, DAB;
        a = sqrt((B.x- A.x) * (B.x - A.x) + (B.y - A.y) * (B.y - A.y));
        b = sqrt((C.x- B.x) * (C.x - B.x) + (C.y - B.y) * (C.y - B.y));
        c = sqrt((C.x- D.x) * (C.x - D.x) + (C.y - D.y) * (C.y - D.y));
        d = sqrt((D.x- A.x) * (D.x - A.x) + (D.y - A.y) * (D.y - A.y));
        d1 = sqrt((B.x- D.x) * (B.x - D.x) + (B.y - D.y) * (B.y - D.y));
        d2 = sqrt((C.x- A.x) * (C.x - A.x) + (C.y - A.y) * (C.y - A.y));
        ABC = (a * a + b * b - d2 * d2) / 2 * a * b;
        BCD = (b * b + c * c - d1 * d1) / 2 * b * c;
        CDA = (d * d + c * c - d2 * d2) / 2 * d * c;
        DAB = (a * a + d * d - d1 * d1) / 2 * a * d;
        if(ABC != BCD || ABC != CDA || ABC != DAB)
            throw std::logic_error("It`s not a rectangle");
    }
    Rectangle<T>() = default;

    double Area() {
        double a =  sqrt((B.x - A.x) * (B.x - A.x) + (B.y - A.y) * (B.y - A.y));
        double b =  sqrt((C.x - B.x) * (C.x - B.x) + (C.y - B.y) * (C.y - B.y));
        return a * b;
    }

    void Print(std::ostream& os) {
        os << A << " " << B << " " << C << " " << D << std::endl;
    }

    void operator<< (std::ostream& os) {
        os << A << " " << B << " " << C << " " << D;
    }
};

#endif
```

stack.h

```
#ifndef OOP_EXERCISE_05_STACK_H
#define OOP_EXERCISE_05_STACK_H

#include <iterator>
#include <memory>
#include <algorithm>

namespace containers {


    template<class T, class Allocator = std::allocator<T>>
    class stack {
    private:
        struct element;
        size_t size = 0;
    public:
        stack() = 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 stack;
        };

        forward_iterator begin();
        forward_iterator end();
        void push(const T& value);
        T& top();
        void pop();
        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);
        stack& operator=(stack& other);
        size_t Size();
    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_;
        };

        struct element {
            T value;
            std::unique_ptr<element, deleter> next_element {nullptr, deleter{nullptr}};
            element(const T& value_): value(value_) {}
            forward_iterator next();
        };
        allocator_type allocator_{};
        std::unique_ptr<element, deleter> first{nullptr, deleter{nullptr}};
    };

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

    template<class T, class Allocator>
    typename stack<T, Allocator>::forward_iterator stack<T, Allocator>::end() {
        return forward_iterator(nullptr);
    }

    template<class T, class Allocator>
    void stack<T, Allocator>::push(const T& value) {
        element* tmp = this->allocator_.allocate(1);
        std::allocator_traits<allocator_type>::construct(this->allocator_, tmp, value);
        if (first == nullptr){
            first = std::unique_ptr<element, deleter> (tmp, deleter{&this->allocator_});
        }else{
            std::swap(tmp->next_element, first);
            first = std::move(std::unique_ptr<element, deleter> (tmp, deleter{&this-
>allocator_}));
        }
        size++;
    }

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

    template<class T, class Allocator>
    T& stack<T, Allocator>::top() {
        if (size == 0) {
            throw std::logic_error ("stack is empty");
        }
        return first->value;
    }

    template<class T, class Allocator>
```

```
   stack<T, Allocator>& stack<T, Allocator>::operator=(stack<T, Allocator>&
other){
      size = other.size;
      first = std::move(other.first);
   }

   template<class T, class Allocator>
   size_t stack<T, Allocator>::Size() {
      return size;
   }

   template<class T, class Allocator>
   void stack<T, Allocator>::delete_by_it(containers::stack<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 stack<T, Allocator>::delete_by_number(size_t 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 stack<T, Allocator>::insert_by_it(containers::stack<T,
Allocator>::forward_iterator ins_it, T& value) {
      auto tmp = std::unique_ptr<element, deleter>(new element{value},
deleter{&this->allocator_});
      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 stack<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);
    }

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

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

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

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

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

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

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


#endif
```

allocator.h

```
#ifndef OOP_EXERCISE_05_ALLOCATOR_H_
#define OOP_EXERCISE_05_ALLOCATOR_H_

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

namespace allocators {

    template<class T, size_t a_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, a_size>;
        };

        my_allocator():
                begin(new char[a_size]),
                end(begin + a_size),
                tail(begin)
        {}

        my_allocator(const my_allocator&) = delete;
        my_allocator(my_allocator&&) = delete;

        ~my_allocator() {
            delete[] begin;
        }

        T* allocate(std::size_t n);
        void deallocate(T* ptr, std::size_t n);

    private:
        char* begin;
        char* end;
        char* tail;
        containers::stack<char*> free_blocks;
    };

    template<class T, size_t a_size>
    T* my_allocator<T, a_size>::allocate(std::size_t n) {
        if (n != 1) {
            throw std::logic_error("can`t allocate arrays");
        }
        if (size_t(end - tail) < sizeof(T)) {
            if (free_blocks.Size()) {
                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*>(tail);
        tail += sizeof(T);
        return result;
    }

    template<class T, size_t a_size>
    void my_allocator<T, a_size>::deallocate(T *ptr, std::size_t n) {
        if (n != 1) {
            throw std::logic_error("can`t allocate arrays");
        }
        if(ptr == nullptr){
            return;
        }
        free_blocks.push(reinterpret_cast<char*>(ptr));
    }
}

#endif
```

CMakeLists.txt

```
cmake_minimum_required (VERSION 3.5)

project(lab6)

add_executable(oop_exercise_06
  main.cpp)

set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -Wextra")

set_target_properties(oop_exercise_06 PROPERTIES CXX_STANDART 14
CXX_STANDART_REQUIRED ON)
```

6. Набор testcases

test_01.txt

```
1
0 0 0 1 6 1 6 0
2
1
-2 0 -2 2 4 2 4 0
5
6
8
7
4
1
5
3
```

```
5
q
```

```
test_02.txt
8
q
```

7. Результаты выполнения тестов

```
walien@PC-name:~/2kurs/CPP/lab6/tmp$ ./oop_exercise_06 <
~/2kurs/CPP/lab6/test_01.txt
choose option (m - man)
choose option (m - man)
enter position to insert
enter rectangle
choose option (m - man)
(-2 0) (-2 2) (4 2) (4 0)
(0 0) (0 1) (6 1) (6 0)
choose option (m - man)
enter max area
1
choose option (m - man)
(-2 0) (-2 2) (4 2) (4 0)
choose option (m - man)
enter position to delete
choose option (m - man)
(0 0) (0 1) (6 1) (6 0)
choose option (m - man)
choose option (m - man)
choose option (m - man)
walien@PC-name:~/2kurs/CPP/lab6/tmp$ ./oop_exercise_06 <
~/2kurs/CPP/lab6/test_02.txt
choose option (m - man)
0 0,  1 1,
8 64,  9 81,
choose option (m - man)
```

8. Объяснение результатов работы программы - вывод

Аллокатор, совместимый со стандартными функциями, описан в allocator.h и используется коллекцией stack.

В ходе данной лабораторной работы были получены навыки работы с аллокаторами. Аллокаторы позволяют ускорить быстроействие программ, сократив количество системных вызовов, а так же усилить контроль над менеджментом памяти.