

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:	Ли А. И.
Группа:	М80-208Б-18
Преподаватель:	Журавлев А.А.
Вариант:	13
Оценка:	
Дата:	

Москва
2019

1. Задание

Разработать программу на языке C++ согласно варианту задания. Программа на C++ должна собираться с помощью системы сборки CMake. Программа должна получать данные из стандартного ввода и выводить данные в стандартный вывод.

Создать шаблон динамической коллекции, согласно варианту задания (треугольник, список, динамический массив):

1. Коллекция должна быть реализована с помощью умных указателей (`std::shared_ptr`, `std::weak_ptr`).

Опционально использование `std::unique_ptr`;

2. В качестве параметра шаблона коллекция должна принимать тип данных;

3. Коллекция должна содержать метод доступа:

Стек – pop, push, top;

Очередь – pop, push, top;

Список, Динамический массив – доступ к элементу по оператору `[]`;

Реализовать аллокатор, который выделяет фиксированный размер памяти (количество блоков памяти – является параметром шаблона аллокатора). Внутри аллокатор должен хранить указатель на используемый блок памяти и динамическую коллекцию указателей на свободные блоки. Динамическая коллекция должна соответствовать варианту задания (Динамический массив, Список, Стек, Очередь);

Коллекция должна использовать аллокатор для выделения и освобождения памяти для своих элементов.

Аллокатор должен быть совместим с контейнерами `std::map` и `std::list` (опционально – `vector`).

Реализовать программу, которая: Позволяет вводить с клавиатуры фигуры (с типом `int` в качестве параметра шаблона фигуры) и добавлять в

коллекцию использующую аллокатор; Позволяет удалять элемент из коллекции по номеру элемента; Выводит на экран введенные фигуры с помощью `std::for_each`;

2. Адрес репозитория на GitHub

https://github.com/elips0n/oop_exercise_06

3. Код программы на C++

main.cpp

```
#include<iostream>
#include<algorithm>
#include<locale.h>
#include"list.h"
#include"allocator.h"
#include"triangle.h"

void Menu1() {
    std::cout << "1. Добавить фигуру в список\n";
    std::cout << "2. Удалить фигуру\n";
    std::cout << "3. Вывести фигуру\n";
    std::cout << "4. Вывести все фигуры с помощью std::for_each()\n";
}

void PushMenu() {
    std::cout << "1. Добавить фигуру в начало списка\n";
    std::cout << "2. Добавить фигуру в конец списка\n";
    std::cout << "3. Добавить фигуру по индексу\n";
}

void DeleteMenu() {
    std::cout << "1. Удалить фигуру в начале списка\n";
    std::cout << "2. Удалить фигуру в конце списка\n";
    std::cout << "3. Удалить фигуру по индексу\n";
}

void PrintMenu() {
    std::cout << "1. Вывести первую фигуру в списке\n";
    std::cout << "2. Вывести последнюю фигуру в списке\n";
    std::cout << "3. Вывести фигуру по индексу\n";
}
```

```
}
```

```
int main() {  
    containers::list<Triangle, allocators::my_allocator<Triangle, 500>> MyList;
```

```
  
    Triangle TempTriangle;  
    uint fc = 1;
```

```
  
    while (true) {  
        Menu1();  
        int n, m, ind;  
        double s;  
        std::cin >> n;  
        switch (n) {  
            case 1:  
                TempTriangle.Read(std::cin);  
                PushMenu();  
                std::cin >> m;  
                switch (m) {  
                    case 1:  
                        MyList.push_front(TempTriangle);  
                        break;  
                    case 2:  
                        MyList.push_back(TempTriangle);  
                        break;  
                    case 3:  
                        std::cout << "Введите индекс позиции: ";  
                        std::cin >> ind;  
                        MyList.insert_by_number(ind, TempTriangle);  
                    default:  
                        break;  
                }  
                break;  
            case 2:  
                DeleteMenu();  
                std::cin >> m;  
                switch (m) {  
                    case 1:  
                        MyList.pop_front();  
                        break;  
                    case 2:  
                        MyList.pop_back();  
                        break;  
                    case 3:  
                        std::cout << "Введите индекс позиции: ";
```

```

        std::cin >> ind;
        MyList.delete_by_number(ind);
        break;
    default:
        break;
    }
    break;
case 3:
    PrintMenu();
    std::cin >> m;
    switch (m) {
    case 1:
        MyList.front().Print();
        std::cout << std::endl;
        break;
    case 2:
        MyList.back().Print();
        std::cout << std::endl;
        break;
    case 3:
        std::cout << "Введите индекс позиции: ";
        std::cin >> ind;
        MyList[ind].Print();
        std::cout << std::endl;
        break;
    default:
        break;
    }
    break;
case 4:
    if (MyList.length() == 0)
    {
        std::cout << "Список пуст.\n" << std::endl;
        break;
    }
    fc = 0;
    std::for_each(MyList.begin(), MyList.end(), [fc](Triangle &X)
mutable {std::cout << "\n Фигура № " << fc << std::endl; X.Print(); std::cout <<
std::endl; fc++; });
    break;
    default:
        return 0;
    }
}

```

```

    system("pause");
    return 0;
}

```

allocator.h

```
#pragma once
```

```

#include <cstdlib>
#include <iostream>
#include <type_traits>
#include <queue>

```

```
namespace allocators {
```

```

    template<class T, size_t ALLOC_SIZE> //ALLOC_SIZE - размер, который
    требуется выделить

```

```
    struct my_allocator {
```

```
    private:
```

```

        char* pool_begin; //указатель на начало хранилища
        char* pool_end; //указатель на конец хранилища
        char* pool_tail; //указатель на конец заполненного пространства
        std::queue<char*> free_blocks;

```

```
    public:
```

```

        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);

};

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.size()) {//ищем свободное место в районе отданном
пространстве
            char* ptr = free_blocks.front();
            free_blocks.pop();
            return reinterpret_cast<T*>(ptr);
        }
        std::cout<<"Bad Alloc"<<std::endl;
        throw std::bad_alloc();
    }
    T* result = reinterpret_cast<T*>(pool_tail);//приведение к типу
    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));
}

}

```

list.h

#pragma once

```
#include <iterator>
#include <memory>
```

```
namespace containers {
```

```
    template<class T, class Allocator = std::allocator<T>>
    class list {
    private:
        struct element; //объявление типа хранящегося в list, для того, чтобы он
        //был виден forward_iterator
        size_t size = 0; //размер списка
    public:
        list() = default; //конструктор по умолчанию

        class forward_iterator {
        public:
            using value_type = T;
            using reference = value_type& ;
            using pointer = value_type* ;
            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 list;
        };

        forward_iterator begin();
        forward_iterator end();
        void push_back(const T& value);
        void push_front(const T& value);
        T& front();
        T& back();
        void pop_back();
        void pop_front();
        size_t length();
        bool empty();
        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);
list& operator=(list& other);
T& operator[](size_t index);
private:
    using allocator_type = typename Allocator::template
rebind<element>::other;

    struct deleter {
    private:
        allocator_type* allocator_;
    public:
        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);
            }
        }
    };

    using unique_ptr = std::unique_ptr<element, deleter>;
    struct element {
        T value;
        unique_ptr next_element = { nullptr, deleter{nullptr} };
        element* prev_element = nullptr;
        element(const T& value_) : value(value_) {}
        forward_iterator next();
    };

    allocator_type allocator_;
    unique_ptr first{ nullptr, deleter{nullptr} };
    element* tail = nullptr;
};

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

template<class T, class Allocator>
typename list<T, Allocator>::forward_iterator list<T, Allocator>::end() {/+

```

```

        return forward_iterator(nullptr);
    }
    template<class T, class Allocator>
    size_t list<T, Allocator>::length() {/+
        return size;
    }
    template<class T, class Allocator>
    bool list<T, Allocator>::empty() {
        return length() == 0;
    }

    template<class T, class Allocator>
    void list<T, Allocator>::push_back(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_ });
        element* temp = tail;/?
        tail = tail->next_element.get();
        tail->prev_element = temp;/?
        size++;
    }

    template<class T, class Allocator>
    void list<T, Allocator>::push_front(const T& value) {
        size++;
        element* result = this->allocator_.allocate(1);
        std::allocator_traits<allocator_type>::construct(this->allocator_, result,
value);
        unique_ptr tmp = std::move(first);
        first = unique_ptr(result, deleter{ &this->allocator_ });
        first->next_element = std::move(tmp);
        if (first->next_element != nullptr)
            first->next_element->prev_element = first.get();
        if (size == 1) {
            tail = first.get();
        }
        if (size == 2) {
            tail = first->next_element.get();
        }
    }

```

```

    }
}

```

```

template<class T, class Allocator>
void list<T, Allocator>::pop_front() {
    if (size == 0) {
        throw std::logic_error("can't pop from empty list");
    }
    if (size == 1) {
        first = nullptr;
        tail = nullptr;
        size--;
        return;
    }
    unique_ptr tmp = std::move(first->next_element);
    first = std::move(tmp);
    first->prev_element = nullptr;
    size--;
}

```

```

template<class T, class Allocator>
void list<T, Allocator>::pop_back() {
    if (size == 0) {
        throw std::logic_error("can't pop from empty list");
    }
    if (tail->prev_element){
        element* tmp = tail->prev_element;
        tail->prev_element->next_element = nullptr;
    }
    else{
        first = nullptr;
        tail = nullptr;
    }
    size--;
}

```

```

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

```

```

template<class T, class Allocator>
T& list<T, Allocator>::back() {
    if (size == 0) {
        throw std::logic_error("list is empty");
    }
    forward_iterator i = this->begin();
    while ( i.it_ptr->next() != this->end()) {
        i++;
    }
    return *i;
}

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

template<class T, class Allocator>
void list<T, Allocator>::delete_by_it(containers::list<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_front();
        return;
    }
    if (d_it.it_ptr == tail) {
        this->pop_back();
        return;
    }

    if (d_it.it_ptr == nullptr) throw std::logic_error("out of broders");
    auto temp = d_it.it_ptr->prev_element;
    unique_ptr temp1 = std::move(d_it.it_ptr->next_element);
    d_it.it_ptr = d_it.it_ptr->prev_element;
    d_it.it_ptr->next_element = std::move(temp1);
    d_it.it_ptr->next_element->prev_element = temp;
    size--;
}

template<class T, class Allocator>
void list<T, Allocator>::delete_by_number(size_t N) {

    if (this->length() == 0)
    {

```

```

        std::cerr << "Нет фигур для удаления. Длина списка 0.\n\n";
        return;
    }
    if (N<0 || N>(this->length()-1)
    {
        std::cerr << "Введенный индекс находится за пределами
возможных значений\n\n";
        return;
    }
    if (N==(this->length() - 1)
    {
        pop_back();
        std::cout << "Фигура удалена из списка.\n" << std::endl;
        return;
    }
    forward_iterator it = this->begin();
    for (size_t i = 0; i < N; ++i) {
        ++it;
    }
    this->delete_by_it(it);
    std::cout << "Фигура удалена из списка.\n" << std::endl;
}

```

```

template<class T, class Allocator>
void list<T, Allocator>::insert_by_it(containers::list<T,
Allocator>::forward_iterator ins_it, T& value) {

    if (ins_it == this->begin()) {
        this->push_front(value);
        return;
    }
    if(ins_it.it_ptr == nullptr){
        this->push_back(value);
        return;
    }
    element* tmp = this->allocator_.allocate(1);
    std::allocator_traits<allocator_type>::construct(this->allocator_, tmp, value);

    forward_iterator i = this->begin();

    tmp->prev_element = ins_it.it_ptr->prev_element;
    ins_it.it_ptr->prev_element = tmp;
    tmp->next_element = std::move(tmp->prev_element->next_element);
    tmp->prev_element->next_element = unique_ptr(tmp, deleter{ &this-
>allocator_ });
}

```

```

        size++;
    }

template<class T, class Allocator>
void list<T, Allocator>::insert_by_number(size_t N, T& value) {
    if (N<0 || N>this->length())
    {
        std::cerr << "Введенный индекс находится за пределами
ВОЗМОЖНЫХ значений\n\n";
        return;
    }
    if (N==0)
    {
        push_front(value);
        return;
    }

    forward_iterator it = this->begin();
    for (size_t i = 0; i < N; ++i) {
        ++it;
    }
    this->insert_by_it(it, value);
}

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

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

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

template<class T, class Allocator>
T& list<T, Allocator>::operator[](size_t index) {
    if (index < 0 || index >= size) {
        throw std::out_of_range("out of list's borders");
    }
}

```

```

        forward_iterator it = this->begin();
        for (size_t i = 0; i < index; i++) {
            it++;
        }
        return *it;
    }

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

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

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

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

```

CMakeLists.txt

```

cmake_minimum_required(VERSION 3.10)
project(oop6)

```

```

set(CMAKE_CXX_STANDARD 17)

```

```

add_executable(main main.cpp)

```

4. Объяснение результатов работы программы

Программа выводит меню, в котором описываются все применимые к фигурам функции – вставка, удаление и вывод фигур из трех различных мест. Функционально программа не изменилась, однако для реализованного ранее списка был написан аллокатор, который более грамотно распоряжается памятью, отведенной для хранения списка фигур.

5. Вывод

В ходе данной лабораторной работы были лучше освоены навыки работы с коллекциями. С помощью пользовательских аллокаторов программист может более эффективно распоряжаться отданной для хранения фигур памятью, сам следить за процессом выделения и очистки памяти, конструирования и деконструирования объектов.