МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСОЙ ФЕДЕРАЦИИ МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЬЕЛЬСКИЙ УНИВЕРСТИТЕТ)

ЛАБОРАТОРНАЯ РАБОТА №6

по курсу объектно-ориентированное программирование I семестр, 2019/20 уч. год

Студент *Артемьев Дмитрий Иванович, группа М8О-206Б-18*

Преподаватель *Журавлёв Андрей Андреевич*

Условие

Задание №1: создать шаблонный класс коллекции стек, по заданию содержащей треугольники, с возможностью

1. push
2. pop
3. top
4. insert
5. erase
6. count_if
7. print

Реализовать аллокатор, который выделяет фиксированный размер памяти. Внутри аллокатор должен хранить указатель на используемый блок памяти и динамическую коллекцию указателей на свободные блоки. Динамическая коллекция в моём варианте - динамический массив.

Описание программы

Исходный код лежит в 8 файлах:

1. src/main.cpp: основная интерактивная программа, возможность работать со стеком
2. include/stack.hpp: описание и реализация класса стека
3. include/figure.hpp: описание и реализация класса обобщённой фигуры
4. include/triangle.hpp: описание и реализация класса треугольника
5. include/template.hpp: разные шаблонные вещи
6. include/point.hpp: описание и реализация класса точки
7. include/allocator.hpp: описание и реализация класса аллокатора
8. include/vector.hpp: описание и реализация класса вектора

Дневник отладки

Я 6 долбанных часов дебажил маленькую описку в методе *erase* класса vector!!!

Недочёты

Двойная линия при вставке и удалении элемента по итератору.

Выводы

Я научился писать аллокаторы памяти для динамических коллекций, использовать библиотеку Boost Test.

Исходный код

allocator.hpp

```
#pragma once

#include <cstdlib>
#include <cstdint>
#include <utility>

#include <exception>
#include <iostream>
#include <type_traits>

#include "vector.hpp"

template<class T, size_t BLOCK_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, BLOCK_SIZE>;
    };

    my_allocator():
        memory_pool_begin_(new char[BLOCK_SIZE]),
        memory_pool_end_(memory_pool_begin_ + BLOCK_SIZE),
        memory_pool_tail_(memory_pool_begin_)
    {}

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

    ~my_allocator() {
        delete[] memory_pool_begin_;
    }

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

```
private:
    size_t search_free_block(size_t n);

    char* memory_pool_begin_;
    char* memory_pool_end_;
    char* memory_pool_tail_;
    bad::TVector<std::pair<char*, size_t>> free_blocks_;
};

template<class T, size_t BLOCK_SIZE>
T* my_allocator<T, BLOCK_SIZE>::allocate(std::size_t n) {

    if(size_t(memory_pool_end_ - memory_pool_tail_) < sizeof(T) * n){
        if(!free_blocks_.empty()){
            size_t idx = search_free_block(n);
            char* ptr = free_blocks_[idx].first;
            free_blocks_.erase(idx);
            return reinterpret_cast<T*>(ptr);
        }
        throw std::bad_alloc();
    }
    else {
        T* result = reinterpret_cast<T*>(memory_pool_tail_);
        memory_pool_tail_ += n * sizeof(T);
        return result;
    }
    throw std::bad_alloc();
}

template<class T, size_t BLOCK_SIZE>
void my_allocator<T, BLOCK_SIZE>::deallocate(T* ptr, std::size_t n) {
    if(ptr == nullptr){
        return;
    }
    char* freePtr = reinterpret_cast<char*>(ptr);
    size_t freeSize = n * sizeof(T);
    free_blocks_.push_back(std::pair{freePtr, freeSize});
}

template<class T, size_t BLOCK_SIZE>
size_t my_allocator<T, BLOCK_SIZE>::search_free_block(size_t n) {
    for (size_t i = 0; i < free_blocks_.size(); i++) {
        if (free_blocks_[i].second >= n * sizeof(T))
            return i;
    }
    throw std::bad_alloc();
}
```

vector.hpp

```cpp
#pragma once

namespace bad {

    template<class T>
    class TVector {
    private:
        size_t size_;
        size_t cap_;
        T *data_;
    public:
        TVector() {
            size_ = 0;
            cap_ = 0;
            data_ = nullptr;
        };

        TVector(int n) {
            size_ = n;
            cap_ = n;
            data_ = new T[cap_];
        }

        TVector(int n, T x){
            size_ = n;
            cap_ = n;
            data_ = new T[cap_];
            for (size_t i = 0; i < size_; i++)
                data_[i] = x;
        }

        TVector(const TVector<T>& other){
            if (this != &other) {
                T* tmp = new T[other.size_];
                std::copy(other.data_, other.data_ + other.size_, tmp);
                data_ = tmp;
                size_ = other.size_;
                cap_ = other.cap_;
            }
        }
```

```
~TVector(){
    delete[] data_;
}

T& operator[] (const int id) const{
    return data_[id];
}

TVector<T>& operator= (const TVector<T>& other) {
    if (this != &other) {
        T* tmp = new T[other.size_];
        std::copy(other.data_, other.data_ + other.size_, tmp);
        delete[] data_;
        data_ = tmp;
        size_ = other.size_;
        cap_ = other.cap_;
    }
    return *this;
}

void erase(const size_t& idx) {
    for (size_t i = idx; i < size_-1; i++) {
        data_[i] = data_[i+1];
    }
    --size_;
}

void push_back(const T& newdata_) {
    if (cap_ == size_){
        cap_ *= 2;
        if (cap_ == 0)
            cap_ = 1;
        T *tmp = new T[cap_];
        for (size_t i = 0; i < size_; ++i) {
            tmp[i] = data_[i];
        }
        delete[] data_;
        data_ = tmp;
    }
    data_[size_++] = newdata_;
}
```

```
        size_t size() const {
            return size_;
        }

        bool empty() {
            return size_ == 0;
        }
    };

} // namespace container
```

figure.hpp

```cpp
#pragma once

#include <iostream>

#include "template.hpp"
#include "point.hpp"

template<typename T, size_t S>
class Figure {
public:
    Figure (std::istream& in) {
        std::cout << "Enter figure (6 ints)" << std::endl;
        for (size_t i = 0; i < S; i++) {
            in >> pts[i];
        }
    }
    int double_area() const;
    Point<double> center() const;
    void print(std::ostream& out) const;

    Point<T>& operator[] (size_t id) {
        return pts[id];
    }
    const Point<T>& operator[] (size_t id) const {
        return pts[id];
    }

    virtual ~Figure() {};
protected:
    Point<T> pts[S];
};

template<typename T, size_t S>
int Figure<T, S>::double_area() const {
    double ans = 0;
    for (size_t i = 1; i < S; i++) {
        ans += crossProd(pts[0], pts[i], pts[i-1]);
    }
    return std::abs(ans);
}
```

```cpp
template<typename T, size_t S>
Point<double> Figure<T, S>::center() const {
    Point<double> ans{0, 0};
    for (size_t i = 1; i < S; i++) {
        ans += pts;
    }
    return ans / (double)(S);
}
```

point.hpp

```cpp
#pragma once

#include <iostream>

template<class T>
struct Point {
    T x, y;

    Point(T x, T y) : x(x), y(y) {};
    Point() = default;
};

template<class T>
std::istream& operator>> (std::istream& in, Point<T>& pt) {
    in >> pt.x >> pt.y;
    return in;
}

template<class T>
std::ostream& operator<< (std::ostream& out, Point<T> const& pt) {
    out << pt.x << ' ' << pt.y;
    return out;
}

template<class T>
Point<T> operator+ (Point<T> const& a, Point<T> const& b) {
    return Point<T>{a.x + b.x, a.y + b.y};
}

template<class T>
Point<T> operator- (Point<T> const& a, Point<T> const& b) {
    return Point<T>{a.x - b.x, a.y - b.y};
}

template<class T>
Point<T> operator* (Point<T> const& a, T const& b) {
    return Point<T>{a.x * b, a.y * b};
}

template<class T>
Point<T> operator/ (Point<T> const& a, T const& b) {
    return Point<T>{a.x / b, a.y / b};
}

template<class T>
bool operator== (Point<T> const& a, Point<T> const& b) {
    return (a.x == b.x) && (a.y == b.y);
}
```

stack.hpp

```cpp
#pragma once

#include <memory>
#include <iterator>

namespace bad {

template<class T, class Allocator = std::allocator<T>>
class stack{
private:
    struct node;

public:
    stack() = 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) : ptr(ptr) {};

        T& operator* ();
        forward_iterator& operator++ ();
        forward_iterator operator++ (int);

        forward_iterator operator+ (int id) {
            if (id < 0) {
                throw 9;
            }
            for (int i = 0; i < id; i++) {
                if (*this == nullptr) {
                    throw 9;
                }
                (*this)++;
            }
            return *this;
        }
        bool operator== (const forward_iterator& o) const;
        bool operator!= (const forward_iterator& o) const;

        //private:

        node *ptr;
        friend class stack;
    };

    forward_iterator begin();
    forward_iterator end();
    size_t size();

    void insert(const forward_iterator& it, const T& value); ///
    void erase(const forward_iterator& it); ///

    void push(const T& value);
    void pop();
    T& top() const;

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>;
    struct node{
        T value;
        unique_ptr nextNode{nullptr,deleter{&this->allocator_}};

        forward_iterator next();

        node(const T &value, unique_ptr next) : value(value), nextNode(std::move(next))
        friend class stack;
    };

    allocator_type allocator_{};
    unique_ptr head{nullptr, deleter{&this->allocator_}};
    size_t size_ = 0;
};


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

template<class T, class Allocator>
typename stack<T, Allocator>::forward_iterator& stack<T, Allocator>::forward_iterator::o
    *this = ptr->next();
    return *this;
}

template<class T, class Allocator>
typename stack<T, Allocator>::forward_iterator stack<T, Allocator>::forward_iterator::op
    forward_iterator prev = *this;
    ++(*this);
    return prev;
}



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

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

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

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

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

template<class T, class Allocator>
typename stack<T, Allocator>::forward_iterator stack<T, Allocator>::node::next() {
    return {nextNode.get()};
}

template<class T, class Allocator>
void stack<T, Allocator>::insert(const forward_iterator& it, const T& value) {
    if (it == this->begin()) {
        this->push(value);
        return;
    }

    forward_iterator tmp = this->begin();
    while (tmp.ptr->next() != it) {
        tmp++;
    }
    node* newptr = this->allocator_.allocate(1);
    std::allocator_traits<allocator_type>::
        construct(this->allocator_, newptr, value,
                  std::unique_ptr<node,deleter>(nullptr, deleter{&this->allocator_}));

    unique_ptr newNode(newptr, deleter{&this->allocator_});
    newNode->nextNode = std::move(tmp.ptr->nextNode);
    tmp.ptr->nextNode = std::move(newNode);
    ++size_;
}

template<class T, class Allocator>
void stack<T, Allocator>::erase(const forward_iterator& it) {
    if (it == nullptr) {
        throw 11;
    }
    if (it == this->begin()) {
        head = std::move(head->nextNode);
        --size_;
        return;
    }

    forward_iterator tmp = this->begin();
    while (tmp.ptr->next() != it) {
        tmp++;
    }

    tmp.ptr->nextNode = std::move(it.ptr->nextNode);
    --size_;
}

template<class T, class Allocator>
void stack<T, Allocator>::push(const T& value) {
    node* newptr = this->allocator_.allocate(1);
    std::allocator_traits<allocator_type>::
        construct(this->allocator_, newptr, value,
                  std::unique_ptr<node,deleter>(nullptr, deleter{&this->allocator_}));
    unique_ptr newNode(newptr, deleter{&this->allocator_});
    newNode->nextNode = std::move(head);
    head = std::move(newNode);

    ++size_;
}

template<class T, class Allocator>
void stack<T, Allocator>::pop() {
    if (size_ == 0) {
        throw 4;
    }
    head = std::move(head->nextNode);
    --size_;
}

template<class T, class Allocator>
T& stack<T, Allocator>::top() const{
    if (size_ == 0) {
        throw 5;
    }
    return head->value;
}

} // namespace bad
```

template.hpp

```
#pragma once

#include <iostream>
#include <tuple>
#include <cmath>

// read tuple
template<size_t I, class... Ts>
struct reader {
    static void read(std::istream& i, std::tuple<Ts...>& t) {
        i >> std::get<I>(t);
        reader<I - 1, Ts...>::read(i, t);
    }
};

template<class... Ts>
struct reader<0, Ts...> {
    static void read(std::istream& i, std::tuple<Ts...>& t) {
        i >> std::get<0>(t);
    }
};

template<class... Ts>
void read(std::istream& i, std::tuple<Ts...>& t) {
    reader<sizeof...(Ts) - 1, Ts...>::read(i, t);
}
// end read tuple

// point functions
template<class T>
T crossProd(Point<T> b, Point<T> a, Point<T> c) {
    return (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
}

template<class T>
T scalarProd(Point<T> b, Point<T> a, Point<T> c) {
    return (b.x - a.x) * (c.x - a.x) + (b.y - a.y) * (c.x - a.x);
}

template<class T>
T dist(Point<T> a, Point<T> b) {
    return sqrt((b.x - a.x) * (b.x - a.x) + (b.y - a.y) * (b.y - a.y));
}
// end point functions
```

triangle.hpp

```
#pragma once

#include <iostream>

#include "point.hpp"
#include "figure.hpp"

template<class T>
class Triangle : public Figure<T, 3> {
public:
    Triangle() = default;
    Triangle(std::istream& in) : Figure<T, 3>(in) {};

    void print(std::ostream& out) const {
        out << "Triangle: " << std::endl;
        for (int i = 0; i < 3; i++) {
            out << Figure<T, 3>::pts[i] << std::endl;
        }
    }
    ~Triangle() {};
};

template<class T>
std::ostream& operator<< (std::ostream& out, const Triangle<T>& tri) {
    tri.print(out);
    return out;
}
```

main.cpp

```cpp
#include <iostream>
#include <algorithm>
#include <cstddef>

#include "stack.hpp"
#include "triangle.hpp"
#include "allocator.hpp"

void help() {
    std::cout << "usage:\n";
    std::cout << " push <elem> \n insert <index> \n pop \n erase <index> \n";
    std::cout << " top \n print \n count_if <max_area> \n exit \n";
}

int main() {
    bad::stack<Triangle<int>, my_allocator<Triangle<int>, 200> > stack_tri;
    while (std::cin) {
        std::string perform;
        std::cin >> perform;

        try {
            if (perform == "push") {
                Triangle<int> tmp(std::cin);
                try {
                    stack_tri.push(tmp);
                }
                catch(std::bad_alloc& e) {
                    std::cout << e.what() << std::endl;
                    continue;
                }
            }
            else if (perform == "insert") {
                size_t index;
                std::cin >> index;
                Triangle<int> tmp(std::cin);
                try {
                    stack_tri.insert(stack_tri.begin() + index, tmp);
                }
                catch(std::bad_alloc& e) {
                    std::cout << e.what() << std::endl;
                    continue;
                }
            }
            else if (perform == "pop") {
                try {
                    stack_tri.pop();
                }
                catch(std::bad_alloc& e) {
                    std::cout << e.what() << std::endl;
                    continue;
                }
            }
            else if (perform == "erase") {
                int index;
                std::cin >> index;
                try {
                    stack_tri.erase(stack_tri.begin() + index);
                }
                catch(std::bad_alloc& e) {
                    std::cout << e.what() << std::endl;
                    continue;
                }
            }
            else if (perform == "top") {
                std::cout << stack_tri.top();
            }
            else if (perform == "print") {
                for (auto& tri : stack_tri) {
                    std::cout << tri;
                }
            }
            else if (perform == "count_if") {
                int area;
                std::cin >> area;
                std::cout << std::count_if(stack_tri.begin(), stack_tri.end(),
                                           [area](const Triangle<int>& tri) {
                                               return tri.double_area() < area * 2;
                                           }) << std::endl;
            }
            else if (perform == "exit") {
                return 0;
            }
            else if (perform == "help") {
```

```
                help();
            }
            else {
                std::cout << "Wrong command\n";
            }
        }
        catch (int n) {
            std::cout << "Caught exception " << std::endl;
            // something
        }
        std::cout << "OK!\n";
    }
}
```