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 *Попов Данила Андреевич, группа М8О-208Б-18*
Преподаватель *Журавлёв Андрей Андреевич*

Условие

Работа с аллокаторами

Реализовать собственный аллокатор на динамическом массиве. Реализовать структуру данных ОЧЕРЕДЬ на собственном аллокаторе.

Дневник отладки

Очень ленивая имплементация CLI.

Недочёты

Аллокатор не копируется, работает только с единичными объектами.

Выводы

Ещё одна странная лабораторная работа, так как пользовательские аллокаторы рассматриваются совсем поверхностно. Как изучение STL, лабораторная себя оправдывает, но не более. Так же в условии задания не упоминается allocator_traits из которого вытекает rebind_traits. Вместо этого предлагается использовать rebind, который может привести к deprecated коду. Не понравилось.

Исходный код

main.cpp

```cpp
#include <iostream>
#include <string>

#include <point.hpp>
#include <polygon.hpp>
#include <queue.hpp>
#include <allocator.hpp>

using rhombus = basic_polygon<point2d, 4>;

auto constexpr prompt = "~> ";

void read_rhombus(std::istream& in, rhombus& r);

struct print_string_at_loop_end
{
    std::string_view s;

    ~print_string_at_loop_end()
    {
        std::cout << s;
    }
};

int main(const int argc, char* argv[])
{
    oop::queue<rhombus, oop::vector_allocator<rhombus, 0x10>> q;

    std::cout << prompt;
    std::string input;
    while (std::cin >> input)
    {
        print_string_at_loop_end printer{ prompt };

        try {
            if (input == "push")
            {
                rhombus r;
                read_rhombus(std::cin, r);
```

```
    q.push(r);
}
else if (input == "top")
{
    rhombus& r = q.top();
    print2d(std::cout, r);
}
else if (input == "pop")
{
    q.pop();
}
else if (input == "insert")
{
    size_t ix;
    rhombus r;
    std::cin >> ix;
    read_rhombus(std::cin, r);

    auto it = q.begin();
    while (ix > 0)
    {
        ++it;
        --ix;
    }
    q.insert(it, r);
}
else if (input == "erase")
{
    size_t ix;
    std::cin >> ix;

    auto it = q.begin();
    while (ix > 0)
    {
        ++it;
        --ix;
    }
    q.erase(it);
}
else if (input == "print")
{
    size_t i = 0;
```

```
                std::for_each(q.begin(), q.end(),
                    [&i](rhombus& r)
                    {
                        std::cout << "[-- " << i << " --]\n\n";
                        print2d(std::cout, r);
                        ++i;
                    }
                );
            }
            else if (input == "less")
            {
                double area;
                std::cin >> area;
                if (area < 0)
                {
                    std::cout << "invalid area" << std::endl;
                    continue;
                }

                for (auto& r : q)
                {
                    if (area2d(r) < area)
                    {
                        print2d(std::cout, r);
                    }
                }
            }
            else if (input == "exit")
            {
                break;
            }
            else
            {
                std::cout << "Unknown command '" << input << "'" << std::endl;
            }
        }
        catch (std::exception & e)
        {
            std::cout << "error: " << e.what() << std::endl;
        }
    }
}
```

```
void read_rhombus(std::istream& in, rhombus& r)
{
    auto constexpr precision = 0.000000001L;
    for (auto& p : r)
    {
        in >> p;
    }
    if (in.fail())
    {
        return;
    }

    constexpr size_t size = rhombus::size();
    const double dist = distance(r[0], r[size - 1]);
    for (size_t i = 0; i < size - 1; i++)
    {
        const double next = distance(r[i], r[i + 1]);
        if (std::abs(dist - next) > precision)
        {
            in.setstate(std::ios::failbit);
            break;
        }
    }
}
```

include/point.hpp

```
#pragma once

#include <iostream>
#include <cstddef>
#include <cmath>

template <typename _Type, size_t _Dimensions>
struct point {
    static_assert(_Dimensions != 0, "can not create 0d point");

    using value_type = _Type;
    using reference = value_type&;
    using const_reference = const value_type&;
    using pointer = value_type*;
    using const_pointer = const value_type*;

    using iterator = pointer;
    using const_iterator = const_pointer;

    value_type dots[_Dimensions];

    [[nodiscard]] value_type& operator[](size_t ix) noexcept {
        return dots[ix];
    }

    [[nodiscard]] const value_type& operator[](size_t ix) const noexcept {
        return const_cast<point&>(*this).operator[](ix);
    }

    [[nodiscard]] iterator begin() noexcept {
        return &dots[0];
    }

    [[nodiscard]] const_iterator begin() const noexcept {
        return const_cast<point&>(*this).begin();
    }

    [[nodiscard]] iterator end() noexcept {
        return &dots[_Dimensions];
    }
```

```
    [[nodiscard]] const_iterator end() const noexcept {
        return const_cast<point&>(*this).end();
    }

    [[nodiscard]] static constexpr size_t size() noexcept {
        return _Dimensions;
    }

    [[nodiscard]] point operator+(const point& other) const {
        point result = *this;

        for (size_t i = 0; i < result.size(); i++) {
            result[i] += other[i];
        }

        return result;
    }

    [[nodiscard]] point operator-(const point& other) const {
        point result = *this;

        for (size_t i = 0; i < result.size(); i++) {
            result[i] -= other[i];
        }

        return result;
    }
};

template <typename Type, size_t _Dims>
std::ostream& operator<<(std::ostream& stream, const point<Type, _Dims>& p) {
    stream << "{ ";
    for (const auto& d : p) {
        stream << d << " ";
    }
    stream << "}";

    return stream;
}

template <typename _Type, size_t _Dims>
std::istream& operator>>(std::istream& stream, point<_Type, _Dims>& p) {
```

```
    for (auto& d : p) {
        stream >> d;
    }

    return stream;
}

// Examples:
using point2d = point<double, 2>;

inline double distance(const point2d& left, const point2d& right) {
    const double x = left[0] - right[0];
    const double y = left[1] - right[1];
    return std::sqrt((x * x) + (y * y));
}
```

include/polygon.hpp

```
#pragma once

#include <cstddef> // size_t
#include <tuple>
#include <type_traits>
#include <istream>
#include <ostream>
#include <stdexcept>

template<typename _T>
auto print2d(std::ostream& stream, const _T& tuple);

/*
    basic_polygon traits
*/
template<typename _Vertex>
struct basic_polygon_traits {
    using vertex          = _Vertex;
    using pointer         = vertex*;
    using const_pointer   = const vertex*;
    using reference       = vertex&;
    using const_reference = const vertex&;

    using iterator       = pointer;
    using const_iterator = const_pointer;
};

/*
    basic_polygon class
    tuple-like
    structured binding is available
*/
template<typename _Vertex, size_t _NumOfPoints>
class basic_polygon
{
    static_assert(_NumOfPoints >= 3, "can not create polygon from points when there are
    using traits = basic_polygon_traits<_Vertex>;

public:
    using vertex          = typename traits::vertex;
    using pointer         = typename traits::pointer;
```

```
using const_pointer   = typename traits::const_pointer;
using reference       = typename traits::reference;
using const_reference = typename traits::const_reference;

using iterator       = typename traits::iterator;
using const_iterator = typename traits::const_iterator;


// constructors
basic_polygon() = default;
explicit basic_polygon(std::istream& stream) {
    for (auto& point : points) {
        stream >> point;
    }
    if (stream.fail()) {
        throw std::runtime_error("bad polygon initialization");
    }
}
explicit basic_polygon(const vertex& v) noexcept {
    for (auto& point : points) {
        point = v;
    }
}



// element getters
reference at(size_t ix) {
    return points[ix];
}
const_reference at(size_t ix) const {
    return const_cast<basic_polygon&>(*this).at(ix);
}

reference operator[](size_t ix) {
    return at(ix);
}
const_reference operator[](size_t ix) const {
    return const_cast<basic_polygon&>(*this)[ix];
}
```

```
// iterators
iterator begin() {
    return &points[0];
}
const_iterator begin() const {
    // cast const to mutable and use non-const begin
    return const_cast<basic_polygon&>(*this).begin();
}

/* NEVER DEREFERENCE */
iterator end() {
    return &points[_NumOfPoints];
}
/* NEVER DEREFERENCE */
const_iterator end() const {
    // cast const to mutable and use non-const end
    return const_cast<basic_polygon&>(*this).end();
};



// structured binding
template<size_t _Ix>
constexpr auto& get() & {
    // check out of bounds
    if constexpr (_Ix < _NumOfPoints) {
        return points[_Ix];
    }
    else {
        // generate compile-time error
        static_assert(_Ix < _NumOfPoints, "ix is out of range");
    }
}

template<size_t _Ix>
constexpr auto const& get() const& {
    // cast const to mutable and use non-const get
    // which does no effect on storage
    return const_cast<basic_polygon&>(*this).get<_Ix>();
}
```

```
    template<size_t _Ix>
    constexpr auto&& get() && {
        // cast lvalue reference to rvalue and return it
        return std::move(this->get<_Ix>());
    }

    static constexpr size_t size() {
        return _NumOfPoints;
    }

    void write(std::ostream& s);

private:
    vertex points[_NumOfPoints];

    template<size_t _Ix, typename _V, size_t _N>
    friend constexpr auto std::get(const basic_polygon<_V, _N>& polygon);
};

// std types specializations for structured binding of basic_polygon
namespace std {
    template<size_t _Ix, typename _Vertex, size_t _NumOfPoints>
    constexpr auto get(const basic_polygon<_Vertex, _NumOfPoints>& polygon) {
        return polygon.points[_Ix];
    }

    template<typename _Vertex, size_t _NumOfPoints>
    struct tuple_size<::basic_polygon<_Vertex, _NumOfPoints>>
        : integral_constant<size_t, _NumOfPoints> {};

    template<size_t _Ix, typename _Vertex, size_t _NumOfPoints>
    struct tuple_element<_Ix, ::basic_polygon<_Vertex, _NumOfPoints>> {
        using type = typename basic_polygon_traits<_Vertex>::vertex;
    };
} // namespace std

template <typename _Vertex, size_t _NumOfPoints>
void basic_polygon<_Vertex, _NumOfPoints>::write(std::ostream& s) {
    print2d(s, *this);
}

namespace detail {
```

```
template<size_t _Off, size_t ... _Ix>
std::index_sequence<(_Off + _Ix)...> add_offset(std::index_sequence<_Ix...>) {
    return {};
}

template<size_t _Off, size_t _N>
auto make_index_sequence_with_offset() {
    return add_offset<_Off>(std::make_index_sequence<_N>{});
}

template<typename _T, size_t... _Ix>
double area2d(const _T& tuple, std::index_sequence<_Ix...>) {
    using vertex = std::remove_const_t<std::remove_reference_t<decltype(std::get<0>(
    static_assert(std::is_same_v<vertex, point2d>, "incorrect type");

    auto constexpr tuple_size = std::tuple_size<_T>{}();
    auto constexpr x = 0;
    auto constexpr y = 1;

    using std::get;

    double result = ((get<_Ix>(tuple)[x] * (get<_Ix + 1>(tuple)[y] - get<_Ix - 1>(tu
    auto constexpr first = 0;
    auto constexpr last = tuple_size - 1;
    result += get<first>(tuple)[x] * (get<first + 1>(tuple)[y] - get<last>(tuple)[y]
    result += get<last>(tuple)[x] * (get<first>(tuple)[y] - get<last - 1>(tuple)[y])
    result /= 2;

    return std::abs(result);
}

template<typename _T, std::size_t... _Ix>
auto center2d(const _T& tuple, std::index_sequence<_Ix...>) {
    using vertex = std::remove_const_t<std::remove_reference_t<decltype(std::get<0>(
    static_assert(std::is_same_v<vertex, point2d>, "incorrect type");

    auto constexpr tuple_size = std::tuple_size<_T>{}();
    auto constexpr x = 0;
    auto constexpr y = 1;

    vertex result = (std::get<_Ix>(tuple) + ...);
    result[x] /= tuple_size;
```

```
        result[y] /= tuple_size;

        return result;
    }

    template<typename _T, std::size_t... _Ix>
    auto print_points2d(std::ostream& out, const _T& tuple, std::index_sequence<_Ix...>)
        auto constexpr tuple_size = std::tuple_size<_T>{}();
        (out << ... << std::get<_Ix>(tuple));
    }
}

template<typename _T>
double area2d(const _T& tuple) {
    auto constexpr tuple_size = std::tuple_size<_T>{}();
    using vertex = std::remove_reference_t<decltype(std::get<0>(tuple))>;
    return detail::area2d(tuple, detail::make_index_sequence_with_offset<1, tuple_size -
}

template<typename _T>
auto center2d(const _T& tuple) {
    auto constexpr tuple_size = std::tuple_size<_T>{}();
    return detail::center2d(tuple, std::make_index_sequence<tuple_size>{});
}

template<typename _T>
auto print2d(std::ostream& stream, const _T& tuple) {
    auto constexpr tuple_size = std::tuple_size<_T>{}();

    using std::endl;

    stream << "\ntype:   ";
    switch (tuple_size) {
    case 4:
        stream << "rhombus" << endl; break;
    case 5:
        stream << "pentagon" << endl; break;
    case 6:
        stream << "hexagon" << endl; break;
    default:
        stream << "unknown" << endl;
    }
```

```
    stream << "center: " << center2d(tuple) << endl
        << "area:   " << area2d(tuple) << endl
        << "points: ";
    detail::print_points2d(stream, tuple, std::make_index_sequence<tuple_size>{});
    stream << endl << endl;
}
```