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-108Б-18, №6
Студент:	Васильева Василиса Евгеньевна
Преподаватель:	Журавлёв Андрей Андреевич
Оценка:	
Дата:	23.12.2019

Москва, 2019

1. Задание

Собрать шаблон динамической коллекции согласно варианту задания.
Вариант 6: Пятиугольник. Стек. Список.

2. Адрес репозитория на GitHub

https://github.com/vasilisavasileva/oop_exercise_06

3. Код программы на C++

Vertex.h

```
#pragma once
#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>
double vector (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 {};
```

Pentagon.h

```
#pragma once
#include<math.h>
#include<stdio.h>
#include<iostream>
#include"vertex.h"

template<class T>
class Pentagon {
public:
	vertex<T> vertices[5];

	Pentagon() = default;
	Pentagon(std::istream& in);
	void Read(std::istream& in);
	double Area() const;
	void Print(std::ostream& os) const;
	friend std::ostream& operator<< (std::ostream& out, const Pentagon<T>& point);
};

	template<class T>
	Pentagon<T>::Pentagon(std::istream& is) {
		for (int i = 0; i < 5; i++) {
			is >> this->vertices[i];
		}
	}

	template<class T>
	double Pentagon<T>::Area() const {
		double Area = 0;
		for (int i = 0; i < 5; i++) {
			Area += (vertices[i].x) * (vertices[(i + 1) % 5].y) - (vertices[(i +
1) % 5].x) * (vertices[i].y);
		}
		Area *= 0.5;
		return abs(Area);
	}

	template<class T>
	void Pentagon<T>::Print(std::ostream& os) const {
		for (int i = 0; i < 5; i++) {
			os << this->vertices[i];
			if (i != 4) {
				os << ',';
			}
		}
		os << std::endl;
	}

	template<class T>
	void Pentagon<T>::Read(std::istream& in) {
		for (int i = 0;i < 5;i++)
```

```
                in >> vertices[i];
        }

        template<class T>
        std::ostream& operator<<(std::ostream& os, const Pentagon<T>& point) {
                for (int i = 0; i < 5; i++) {
                        os << point.vertices[i];
                        if (i != 5) {
                                os << ',';
                        }
                }
        }
```

Stack.h

```
#pragma once
#include <iterator>
#include <memory>

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();
                T& bottom();
                void pop();
                size_t length();
                void delete_by_it(forward_iterator d_it);
                void delete_by_index(size_t N);
                void insert_by_it(forward_iterator ins_it, T& value);
                void insert_by_index(size_t N, T& value);
                stack& operator=(stack& other);

        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>;
                unique_ptr push_impl(unique_ptr  cur, const T& value);
                unique_ptr pop_impl(unique_ptr cur);

                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;
        };
        /////////
        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>
        size_t stack<T, Allocator>::length() {
                return size;
        }


        template<class T, class Allocator>
        void stack<T,  Allocator>::push(const T& value) {
                first = push_impl(std::move(first), value);
                size++;
        }

        template<class T, class Allocator>
        typename stack<T, Allocator>::unique_ptr stack<T, Allocator>::push_impl(unique_ptr
cur, const T& value) {

                if (cur != nullptr) {
                        cur->next_element = push_impl(std::move(cur->next_element), value);
                        return cur;
                }
                element* result = this->allocator_.allocate(1);
                std::allocator_traits<allocator_type>::construct(this->allocator_, result,
value);
                return unique_ptr(result, deleter{&this->allocator_});
        }
```

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


	template<class T, class Allocator>
	typename stack<T, Allocator>::unique_ptr stack<T, Allocator>::pop_impl(unique_ptr
cur) {
		if (cur->next_element != nullptr) {
			cur->next_element = pop_impl(std::move(cur->next_element));
			return cur;
		}
		return unique_ptr(nullptr, deleter{ &this->allocator_ });//?
	}

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

	template<class T, class Allocator>
	T& stack<T, Allocator>::bottom() {
		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>
	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()) {
//			unique_ptr tmp;
			//element* result = this->allocator_.allocate(1);
			//std::allocator_traits<allocator_type>::construct(this->allocator_,
result, value);
			//return unique_ptr(result, deleter{ &this->allocator_ });
			auto tmp = std::move(first->next_element);
			first = std::move(tmp);
			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_index(size_t N) {
		forward_iterator it = this->begin();
		for (size_t i = 0; i < N; ++i) {
			++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) {
		element* tmp = this->allocator_.allocate(1);
		std::allocator_traits<allocator_type>::construct(this->allocator_, tmp,
value);
		forward_iterator i = this->begin();
		if (ins_it == this->begin()) {
			tmp->next_element = std::move(first);
			first = unique_ptr(tmp, deleter{ &this->allocator_ });
			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 = unique_ptr(tmp, deleter{ &this->allocator_ });
		size++;
	}

	template<class T, class Allocator>
	void stack<T, Allocator>::insert_by_index(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");
		*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;
	}

}
```

Allocator.h

```
#include <cstdlib>
#include <iostream>
#include <type_traits>
#include <list>
#include "Stack.h"

namespace allocators {

	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;
		std::list<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.size()) {
				auto it = free_blocks.begin();
				char* ptr = *it;
				free_blocks.pop_front();
				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_back(reinterpret_cast<char*>(ptr));
	}
};
```

main.cpp

```
#include<iostream>
#include<algorithm>
#include<locale.h>
#include"Pentagon.h"
#include"stack.h"
#include"Allocator.h"

void Menu1() {
	std::cout << "1. Добавить фигуру в стек\n";
	std::cout << "2. Удалить фигуру\n";
	std::cout << "3. Вывести фигуру\n";
	std::cout << "4. Вывести все фигуры\n";
	std::cout << "5. Добавить фигуру по индексу\n";
}

void DeleteMenu() {
	std::cout << "1. Удалить фигуру в вершине стека\n";
	std::cout << "2. Удалить фигуру по индексу\n";
}

void PrintMenu() {
```

```
	std::cout << "1. Вывести первую фигуру в стеке\n";
	std::cout << "2. Вывести последнюю фигуру в стеке\n";
}

int main() {
	setlocale(LC_ALL, "rus");
	containers::stack<Pentagon<int>, allocators::my_allocator<Pentagon<int>, 1000>>
Mystack;

	Pentagon<int> TempPentagon;

	while (true) {
		Menu1();
		int n, m, in;
		size_t ind;
		double s;
		std::cin >> n;
		switch (n) {
		case 1:
			TempPentagon.Read(std::cin);
			TempPentagon.Print(std::cout);
			Mystack.push(TempPentagon);
			break;
		case 2:
			DeleteMenu();
			std::cin >> m;
			switch (m) {
			case 1:
				Mystack.pop();
				break;
			case 2:
				std::cin >> ind;
				Mystack.delete_by_index(ind);
				break;
			default:
				break;
			}
			break;
		case 3:
			PrintMenu();
			std::cin >> m;
			switch (m) {
			case 1:
				Mystack.bottom().Print(std::cout);
				std::cout << std::endl;
				break;
			case 2:
				Mystack.top().Print(std::cout);
				std::cout << std::endl;
				break;
			default:
				break;
			}
			break;
		case 4:
			std::for_each(Mystack.begin(), Mystack.end(), [](Pentagon<int>& X) {
X.Print(std::cout); std::cout << std::endl; });
			break;
		case 5:
			std::cout << "Введите индекс\n";
			std::cin >> ind;
			std::cout << "Введите координаты пентагона\n";
			TempPentagon.Read(std::cin);
			Mystack.insert_by_index(ind, TempPentagon);
			break;
```

```
                default:
                        return 0;
                }
        }
        system("pause");
        return 0;
}
```

4. Объяснение результатов работы программы

Выводящееся меню предлагает пользователю добавить в стек фигуру, вывести все, а также удалить фигуру по индексу или с верхушки стека. Вершинами фигуры являются структуры vertex. В классе фигуры определены такие методы, как считывание координат с потока ввода, подсчет площади, вывод координат на экран. В классе стека описан forward_iterator, при помощи которого осуществляются передвижение и доступ к элементам стека. В классе присутствуют такие методы, как вставка и удаление элемента в текущем месте нахождения итератора, а также вставка и удаление элемента с верхушки стека. В соответствии с выбором пользователя приводится в исполнение определенный метод класса стек. Память для стека выделяется классом аллокатора, который выделяет память сразу под 800 фигур, чтобы сократить время на вызовы для нового выделения.

5. Вывод

Аллокатор это такой класс, который позволяет нам, по сути, вручную управлять выделением памяти и контролировать этот процесс. Он разом выделяет большой объем памяти, а потом «отщипывает» от него по кусочку для заполнения. Это сокращает количество системных вызовов, запрашивающих новые области памяти, которые занимают много времени. При использовании аллокаторов памяти мы делаем нашу программу более производительной.