Московский авиационный институт

(Национальный исследовательский университет)

Факультет: «Информационные технологии и прикладная математика»
Кафедра: 806 «Вычислительная математика и программирование»
Дисциплина: «Объектно-ориентированное программирование»

Лабораторная работа № 6

Тема: Основы работы с памятью. Аллокаторы

Студент: Семенов Илья

Преподаватель: Журавлев А.А.

Дата:

Оценка:

Москва, 2019

1. Постановка задачи

- Разработать шаблон контейнера(реализованный с помощью умных указателей), использующий аллокатор для выделения памяти, способный хранить экземпляры шаблонного класса определенной фигуры. Коллекция должна иметь свои итераторы.

Вариант задания 20:

- Фигура - Трапеция
- Коллекция – Очередь
- Аллокатор основан на очереди

2. Репозиторий github

https://github.com/ilya89099/oop_exercise_06/

3. Описание программы

Реализован шаблонный класс очереди. Данные хранятся с помощью shared_ptr и weak_ptr. Также реализованы классы для обычного и константного итератора, содержащие weak_ptr на узел очереди. Очередь содежит барьерный элемент для упрощения функций вставки, удаления и итерирования. Коллекция может также работать со стандартными алгоритмами. Так же релизован аллокатор, выделяющий определенное шаблонным параметром количество памяти. Аллокатор содержит внутри очередь доступных узлов.

4. Набор testcases

test_01.test

```
add 0 1 1 1 1 1 1 1 1

add 0 2 2 2 2 2 2 2 2

add 1 3 3 3 3 3 3 3 3

add 1 4 4 4 4 4 4 4 4

count 2

print

flex

erase 0
erase 1

print
```

test_01.result

```
Trapeze p1:1 1, p2:1 1, p3:1 1, p4:1 1

Trapeze p1:2 2, p2:2 2, p3:2 2, p4:2 2

Trapeze p1:3 3, p2:3 3, p3:3 3, p4:3 3

Trapeze p1:4 4, p2:4 4, p3:4 4, p4:4 4

0

Trapeze p1:2 2, p2:2 2, p3:2 2, p4:2 2 Trapeze p1:4 4, p2:4 4, p3:4 4, p4:4 4 Trapeze p1:3 3, p2:3 3, p3:3 3, p4:3 3 Trapeze p1:1 1, p2:1 1, p3:1 1, p4:1 1

Incorrect command

Trapeze p1:4 4, p2:4 4, p3:4 4, p4:4 4 Trapeze p1:1 1, p2:1 1, p3:1 1, p4:1 1
```

test_02.test

```
add 0 1 1 1 1 1 1 1 1

add 0 2 2 2 2 2 2 2 2

add 0 3 3 3 3 3 3 3 3

add 0 4 4 4 4 4 4 4 4

add 0 5 5 5 5 5 5 5 5

add 0 6 6 6 6 6 6 6 6

add 0 7 7 7 7 7 7 7 7

add 0 8 8 8 8 8 8 8 8

add 0 9 9 9 9 9 9 9 9

add 0 1 1 1 1 1 1 1 1

add 0 2 2 2 2 2 2 2 2

add 0 3 3 3 3 3 3 3 3

add 0 4 4 4 4 4 4 4 4

add 0 5 5 5 5 5 5 5 5

add 0 6 6 6 6 6 6 6 6

add 0 7 7 7 7 7 7 7 7
```

test_02.result

```
Trapeze p1:1 1, p2:1 1, p3:1 1, p4:1 1

Trapeze p1:2 2, p2:2 2, p3:2 2, p4:2 2

Trapeze p1:3 3, p2:3 3, p3:3 3, p4:3 3

Trapeze p1:4 4, p2:4 4, p3:4 4, p4:4 4

Trapeze p1:5 5, p2:5 5, p3:5 5, p4:5 5

Trapeze p1:6 6, p2:6 6, p3:6 6, p4:6 6

Trapeze p1:7 7, p2:7 7, p3:7 7, p4:7 7

Trapeze p1:8 8, p2:8 8, p3:8 8, p4:8 8

Trapeze p1:9 9, p2:9 9, p3:9 9, p4:9 9

Trapeze p1:1 1, p2:1 1, p3:1 1, p4:1 1

Trapeze p1:2 2, p2:2 2, p3:2 2, p4:2 2

Trapeze p1:3 3, p2:3 3, p3:3 3, p4:3 3

Trapeze p1:4 4, p2:4 4, p3:4 4, p4:4 4

Trapeze p1:5 5, p2:5 5, p3:5 5, p4:5 5

std::bad_alloc

std::bad_alloc
```

5. Результаты выполнения тестов

Все тесты завершились успешно

6. Листинг программы

main.cpp

```
#include <iostream>
#include <map>
#include <string>
#include <algorithm>
#include <tuple>
#include <list>
#include "Trapeze.h"
#include "Queue.h"
#include "Allocator.h"
int main() {
    std::string command;
    Containers::Queue<Trapeze<int>, Allocator<Trapeze<int>,1000>> figures;
    while (std::cin >> command) {
        if (command == "add") {
            size_t position;
            std::cin >> position;
            auto it = figures.begin();
            try {
                it = std::next(it, position);
            } catch(std::exception& e) {
                std::cout << "Position is too big\n";
                continue;
            }
            Trapeze<int> new_figure;
            try {
                std::cin >> new_figure;
                figures.Insert(it, new_figure);
                std::cout << new_figure << "\n";
            } catch (std::exception& ex) {
                std::cout << ex.what() << "\n";
            }
        } else if (command == "erase") {
            size_t index;
            std::cin >> index;
            try {
                auto it = std::next(figures.begin(), index);
                figures.Erase(it);
            } catch (...) {
                std::cout << "Index is too big\n";
                continue;
            }
        } else if (command == "size") {
          std::cout << figures.Size() << "\n";
        } else if (command == "print") {
            std::for_each(figures.begin(), figures.end(), [] (const
Trapeze<int>& fig) {
                std::cout << fig << " ";
            });
            std::cout << "\n";
        } else if (command == "count") {
            size_t required_area;
            std::cin >> required_area;
            std::cout << std::count_if(figures.begin(), figures.end(),
[&required_area] (const Trapeze<int>& fig) {
                return fig.Area() < required_area;
            });
            std::cout << "\n";
        } else {
            std::cout << "Incorrect command" << "\n";
            std::cin.ignore(32767, '\n');
        }
    }
}
```

Allocator.h

```cpp
#pragma once
#include <memory>
#include "Queue.h"
template <typename T, size_t ALLOC_SIZE>
class Allocator {
public:
    using value_type = T;
    using size_type = std::size_t;
    using difference_type = std::ptrdiff_t;
    using is_always_equal = std::false_type;
    Allocator(const Allocator&) = delete;
    Allocator(Allocator&&) = delete;
    template<class V>
    struct rebind {
        using other = Allocator<V, ALLOC_SIZE>;
    };
    Allocator() {
        size_t object_count = ALLOC_SIZE / sizeof(T);
//        std::cout << "alloc size:" << ALLOC_SIZE << "\n";
//        std::cout << "object count:" << object_count << "\n";
//        std::cout << "type size:" << sizeof(T) << "\n";
        memory = reinterpret_cast<char*>(operator new(sizeof(T) *
object_count));
        for (size_t i = 0; i < object_count; ++i) {
            free_blocks.Push(memory + sizeof(T) * i);
        }
    }
    ~Allocator() {
        operator delete(memory);
    }
    T* allocate(size_t size) {
        if (size > 1) {
            throw std::logic_error("This allocator cant do that");
        }
        if (free_blocks.Empty()) {
            throw std::bad_alloc();
        }
        T* temp = reinterpret_cast<T*>(free_blocks.Top());
        free_blocks.Pop();
        return temp;
    }
    void deallocate(T* ptr, size_t size) {
        if (size > 1) {
            throw std::logic_error("This allocator cant do that");
        }
        free_blocks.Push(reinterpret_cast<char*>(ptr));
    }
private:
    Containers::Queue<char*> free_blocks;
    char* memory;
};
```

Queue.h

```
#pragma once
#include <memory>
#include <exception>
namespace Containers {
    template <typename T, typename Allocator>
    class Queue;
    template <typename T>
    class QueueNode;
    template <typename T, typename Allocator>
    class QueueConstIterator;
    template <typename T, typename Allocator>
    class QueueIterator;
    //Implementation of QueueNode
    template <typename T>
    struct QueueNode {
        QueueNode() = default;
        QueueNode(T new_value) : value(new_value) {}
        T value;
        std::shared_ptr<QueueNode> next = nullptr;
        std::weak_ptr<QueueNode> prev;
    };
    //Implementation of Queue
    template<typename T, typename Allocator = std::allocator<T>>
    class Queue {
        friend QueueIterator<T, Allocator>;
        friend QueueConstIterator<T, Allocator>;
        using allocator_type = typename Allocator::template
rebind<QueueNode<T>>::other;
        struct deleter {
            deleter(allocator_type* allocator) : allocator_(allocator) {}
            void operator() (QueueNode<T>* ptr) {
                if (ptr != nullptr) {

std::allocator_traits<allocator_type>::destroy(*allocator_,ptr);
                    allocator_->deallocate(ptr, 1);
                }
            }
        private:
            allocator_type* allocator_;
        };
    public:
        Queue() {
            QueueNode<T>* ptr = allocator_.allocate(1);
            std::allocator_traits<allocator_type>::construct(allocator_,
ptr);
            std::shared_ptr<QueueNode<T>> new_elem(ptr,
deleter(&allocator_));
            tail = new_elem;
            head = tail;
        }
        Queue(const Queue& q) = delete;
        Queue& operator = (const Queue&) = delete;
        void Pop() {
            if (Empty()) {
                throw std::out_of_range("Pop from empty queue");
            }
            head = head->next;
        }
        const T& Top() const {
            return head->value;
        }
        T& Top() {
            return head->value;
        }
        size_t Size() const {
            size_t size = 0;
            for (auto i : *this) {
                size++;
            }
            return size;
        }
        void Push(const T &value) {
            QueueNode<T>* ptr = allocator_.allocate(1);
            std::allocator_traits<allocator_type>::construct(allocator_,
ptr, value);
            std::shared_ptr<QueueNode<T>> new_elem(ptr,
deleter(&allocator_));
            if (Empty()) {
                head = new_elem;
                head->next = tail;
                tail->prev = head;
            } else {
                tail->prev.lock()->next = new_elem;
                new_elem->prev = tail->prev;
                new_elem->next = tail;
                tail->prev = new_elem;
            }
        }
        bool Empty() const {
            return head == tail;
        }
        QueueConstIterator<T, Allocator> begin() const {
            return QueueConstIterator<T, Allocator>(head, this);
        }
        QueueConstIterator<T, Allocator> end() const {
            return QueueConstIterator<T, Allocator>(tail, this);
        }
        QueueIterator<T, Allocator> begin() {
            return QueueIterator<T,Allocator>(head, this);
        }
        QueueIterator<T, Allocator> end() {
            return QueueIterator<T, Allocator>(tail, this);
        }
        void Erase(QueueIterator<T, Allocator> it) {
            if (it.collection != this) {
                throw std::runtime_error("Iterator does not belong to this
collection");
            }
            std::shared_ptr<QueueNode<T>> it_ptr = it.node.lock();
            if (!it_ptr) {
                throw std::runtime_error("Iterator is corrupted");
            }
            if (it == end()) {
                throw std::runtime_error("Erase of end iterator");
            }
            if (it == begin()) {
                Pop();
            } else {
                std::weak_ptr<QueueNode<T>> prev_ptr = it_ptr->prev;
                std::shared_ptr<QueueNode<T>> next_ptr = it_ptr->next;
                prev_ptr.lock()->next = next_ptr;
                next_ptr->prev = prev_ptr;
            }
        }
        void Insert(QueueIterator<T, Allocator> it, const T& value) {
            if (it.collection != this) {
                throw std::runtime_error("Iterator does not belong to this
collection");
            }
            std::shared_ptr<QueueNode<T>> it_ptr = it.node.lock();
            if (!it_ptr) {
                throw std::runtime_error("Iterator is corrupted");
            }
            if (it == end()) {
                Push(value);
                return;
            }
            QueueNode<T>* ptr = allocator_.allocate(1);
            std::allocator_traits<allocator_type>::construct(allocator_,
ptr, value);
            std::shared_ptr<QueueNode<T>> new_elem(ptr,
deleter(&allocator_));
            if (it == begin()) {
                new_elem->next = head;
                head->prev = new_elem;
                head = new_elem;
            } else {
                std::shared_ptr<QueueNode<T>> next_ptr = it_ptr;
                std::weak_ptr<QueueNode<T>> prev_ptr = it_ptr->prev;
                new_elem->prev = prev_ptr;
                prev_ptr.lock()->next = new_elem;
                new_elem->next = next_ptr;
                next_ptr->prev = new_elem;
            }
        }
    private:
        allocator_type allocator_;
        std::shared_ptr<QueueNode<T>> head;
        std::shared_ptr<QueueNode<T>> tail;
    };
    template<typename T, typename Allocator>
    class QueueIterator {
        friend Queue<T, Allocator>;
    public:
        using value_type = T;
        using reference = T&;
        using pointer = T*;
        using difference_type = ptrdiff_t;
        using iterator_category = std::forward_iterator_tag;
        QueueIterator(std::shared_ptr<QueueNode<T>> init_ptr,const Queue<T,
Allocator>* ptr) : node(init_ptr), collection(ptr) {}
        QueueIterator(const QueueIterator& other) : node(other.node),
collection(other.collection) {}
        QueueIterator& operator = (const QueueIterator& other) {
            node = other.node;
            return *this;
        }
        bool operator == (const QueueIterator& other) const {
            auto lhs_l = node.lock(), rhs_l = other.node.lock();
            if (lhs_l && rhs_l) {
                return lhs_l.get() == rhs_l.get();
            }
            return false;
        }
        bool operator != (const QueueIterator& other) const {
            return !(*this == other);
        }
        QueueIterator& operator++() { // prefix
            std::shared_ptr<QueueNode<T>> temp = node.lock();
            if (temp) {
                if (temp->next == nullptr) {
                    throw std::out_of_range("Going out of container
boundaries");
                }
                temp = temp->next;
                node = temp;
                return *this;
            } else {
                throw std::runtime_error("Element pointed by this iterator
doesnt exist anymore");
            }
        }
        QueueIterator operator++(int) { //postfix
            QueueIterator result(*this);
            ++(*this);
            return result;
        }
        T& operator* () const {
            std::shared_ptr<QueueNode<T>> temp = node.lock();
            if (temp) {
                if (temp->next == nullptr) {
                    throw std::runtime_error("Dereferencing of end
iterator");
                }
                return temp->value;
            } else {
                throw std::runtime_error("Element pointed by this iterator
doesnt exist anymore");
            }
        }
    private:
        std::weak_ptr<QueueNode<T>> node;
        const Queue<T, Allocator>* collection;
    };
    template<typename T, typename Allocator>
    class QueueConstIterator {
        friend Queue<T, Allocator>;
    public:
        using value_type = T;
        using reference = T&;
        using pointer = T*;
        using difference_type = ptrdiff_t;
        using iterator_category = std::forward_iterator_tag;
        QueueConstIterator(std::shared_ptr<QueueNode<T>> init_ptr, const
Queue<T, Allocator>* ptr) : node(init_ptr), collection(ptr) {}
        QueueConstIterator(const QueueConstIterator& other) :
node(other.node), collection(other.collection) {}
        QueueConstIterator& operator = (const QueueConstIterator& other) {
            node = other.node;
            return *this;
        }
        bool operator == (const QueueConstIterator& other) const {
            auto lhs_l = node.lock(), rhs_l = other.node.lock();
            if (lhs_l && rhs_l) {
                return lhs_l.get() == rhs_l.get();
            }
            return false;
        }
        bool operator != (const QueueConstIterator& other) const {
            return !(*this == other);
        }
        QueueConstIterator& operator++() { // prefix
            std::shared_ptr<QueueNode<T>> temp = node.lock();
            if (temp) {
                if (temp->next == nullptr) {
                    throw std::out_of_range("Going out of container
boundaries");
                }
                temp = temp->next;
                node = temp;
                return *this;
            } else {
                throw std::runtime_error("Element pointed by this iterator
doesnt exist anymore");
            }
        }
        QueueConstIterator operator++(int) { //postfix
            QueueConstIterator result(*this);
            (*this)++;
            return result;
        }
        const T& operator* () const {
            std::shared_ptr<QueueNode<T>> temp = node.lock();
            if (temp) {
                if (temp->next == nullptr) {
                    throw std::runtime_error("Dereferencing of end
iterator");
                }
                return temp->value;
            } else {
                throw std::runtime_error("Element pointed by this iterator
doesnt exist anymore");
            }
        }
    private:
        std::weak_ptr<QueueNode<T>> node;
        const Queue<T, Allocator>* collection;
    };
}
```

Trapeze.h

```
#pragma once
#include <iostream>
#include <exception>
#include "Point.h"
template <typename T>
class Trapeze {
public:
    Trapeze() = default;
    Trapeze(Point<T> p1, Point<T> p2, Point<T>p3, Point<T> p4);
    Point<T> Center() const;
    double Area() const;
    void Print(std::ostream& os) const;
    void Scan(std::istream& is);
private:
    Point<T> p1_, p2_, p3_, p4_;
};
```

```
template <typename T>
Trapeze<T>::Trapeze(Point<T> p1, Point<T> p2, Point<T> p3, Point<T> p4)
        :  p1_(p1), p2_(p2), p3_(p3), p4_(p4){
    Vector<T> v1(p1_, p2_), v2(p3_, p4_);
    if (v1 = Vector<T>(p1_, p2_), v2 = Vector<T>(p3_, p4_), is_parallel(v1,
v2)) {
        if (v1 * v2 < 0) {
            std::swap(p3_, p4_);
        }
    } else if (v1 = Vector<T>(p1_, p3_), v2 = Vector<T>(p2_, p4_),
is_parallel(v1, v2)) {
        if (v1 * v2 < 0) {
            std::swap(p2_, p4_);
        }
        std::swap(p2_, p3_);
    } else if (v1 = Vector<T>(p1_, p4_), v2 = Vector<T>(p2_, p3_),
is_parallel(v1, v2)) {
        if (v1 * v2 < 0) {
            std::swap(p2_, p3_);
        }
        std::swap(p2_, p4_);
        std::swap(p3_, p4_);
    } else {
        throw std::logic_error("At least 2 sides of trapeze must be
parallel");
    }
}
template <typename T>
Point<T> Trapeze<T>::Center() const {
    return (p1_ + p2_ + p3_ + p4_) / 4;
}
template<typename T>
double Trapeze<T>::Area() const {
    double height = point_and_line_distance(p1_, p3_, p4_);
    return (Vector<T>(p1_, p2_).length() + Vector<T>(p3_, p4_).length()) *
height / 2;
}
template<typename T>
void Trapeze<T>::Print(std::ostream& os) const {
    os << "Trapeze p1:" << p1_ << ", p2:" << p2_ << ", p3:" << p3_ << ",
p4:" << p4_;
}
template <typename T>
void Trapeze<T>::Scan(std::istream &is) {
    Point<T> p1,p2,p3,p4;
    is >> p1 >> p2 >> p3 >> p4;
    *this = Trapeze(p1,p2,p3,p4);
}
template <typename T>
std::ostream& operator << (std::ostream& os, const Trapeze<T>& trap) {
    trap.Print(os);
    return os;
}
template <typename T>
std::istream& operator >> (std::istream& is, Trapeze<T>& trap) {
    trap.Scan(is);
    return is;
}
```

Point.h

```
#pragma once
#include <numeric>
#include <iostream>
#include <vector>
#include <cmath>
#include <limits>
template <typename T>
struct Point {
    T x = 0;
    T y = 0;
};
template <typename T>
class Vector {
public:
    explicit Vector(T a, T b);
    explicit Vector(Point<T> a, Point<T> b);
    bool operator == (Vector rhs);
    Vector operator - ();
    double length() const;
    T x;
    T y;
};
template <typename T>
Point<T> operator + (Point<T> lhs, Point<T> rhs) {
    return {lhs.x + rhs.x, lhs.y + rhs.y};
}
template <typename T>
Point<T> operator - (Point<T> lhs, Point<T> rhs) {
    return {lhs.x - rhs.x, lhs.y - rhs.y};
}
template <typename T>
Point<T> operator / (Point<T> lhs, double a) {
    return { lhs.x / a, lhs.y / a};
}
template <typename T>
Point<T> operator * (Point<T> lhs, double a) {
    return {lhs.x * a, lhs.y * a};
}
template <typename T>
bool operator < (Point<T> lhs, Point<T> rhs) {
    return (lhs.x * lhs.x + lhs.y * lhs.y) < (lhs.x * lhs.x + lhs.y *
lhs.y);
}
template <typename T>
```

```
double operator * (Vector<T> lhs, Vector<T> rhs) {
    return lhs.x * rhs.x + lhs.y * rhs.y;
}
template <typename T>
bool is_parallel(const Vector<T>& lhs, const Vector<T>& rhs) {
    return (lhs.x * rhs.y - lhs.y * rhs.y) == 0;
}
template <typename T>
bool Vector<T>::operator == (Vector<T> rhs) {
    return
            std::abs(x - rhs.x) < std::numeric_limits<double>::epsilon() *
100
            && std::abs(y - rhs.y) < std::numeric_limits<double>::epsilon()
* 100;
}
template <typename T>
double Vector<T>::length() const {
    return sqrt(x*x + y*y);
}
template <typename T>
Vector<T>::Vector(T a, T b)
        : x(a), y(b) {
}
template <typename T>
Vector<T>::Vector(Point<T> a, Point<T> b)
        : x(b.x - a.x), y(b.y - a.y){
}
template <typename T>
Vector<T> Vector<T>::operator - () {
    return Vector(-x, -y);
}
template <typename T>
bool is_perpendecular(const Vector<T>& lhs, const Vector<T>& rhs) {
    return (lhs * rhs) == 0;
}
template <typename T>
double point_and_line_distance(Point<T> p1, Point<T> p2, Point<T> p3) {
    double A = p2.y - p3.y;
    double B = p3.x - p2.x;
    double C = p2.x*p3.y - p3.x*p2.y;
    return (std::abs(A*p1.x + B*p1.y + C) / std::sqrt(A*A + B*B));
}
template <typename T>
std::ostream& operator << (std::ostream& os, const Point<T>& p) {
    return os << p.x << " " << p.y;
}
template <typename T>
std::istream& operator >> (std::istream& is, Point<T>& p) {
    return is >> p.x >> p.y;
}
```

7. Вывод

Выполняя данную лабораторную работу, я получил опыт работы с аллокаторами(структурами, позволяющими более эффективно работать с памятью) и умными указателями. Узнал о применении аллокаторов и научился создавать контейнеры, их использующие.