Московский Авиационный Институт

(Национальный исследовательский Университет)

Факультет: «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

Лабораторной работе № 06
по курсу «Объектно-ориентированное программирование»

Тема:
«Основы работы с коллекциями : итераторы»

Студент:	Пшеницын А. А.
Группа:	М80-208Б-18
Преподаватель:	Журавлев А. А.
Вариант:	17
Оценка:	
Дата:	

Москва
2019

Цель:

Изучение основ работы с контейнерами, знакомство концепцией аллокаторов памяти.
Создать шаблон динамической коллекцию, согласно варианту задания:

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: Динамический массив – доступ к элементу по оператору [];
4. 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);
5. Коллекция должна использовать аллокатор для выделелеления и освобождения памяти для своих элементов.
6. Аллокатор должен быть совместим с контейнерами std::map и std::list (опционально – vector).
7. 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;

Код

main.cpp

```
#include <map>

#include "triangle.h"
#include "que.h"
#include "allocator.h"

int main() {
    int pos;
    queue<triangle<double>, my_allocator<triangle<double>, 10000>> q;
    for (;;) {
        std::string comm;
        std::cin >> comm;
        if (comm == "add") {
            triangle<double> tr(std::cin);
            q.push(tr);
        } else if (comm == "print_top") {
            q.top().print(std::cout);
        } else if (comm == "pop") {
            q.pop();
        } else if (comm == "insert"){
            triangle<double> tr(std::cin);
            std::cin >> pos;
             q.insert_to_num(pos, tr);
        } else if (comm == "erase") {
            std::cin >> pos;
            q.erase_to_num(pos);
        } else if (comm == "print_all") {
```

```
            std::for_each(q.begin(), q.end(), [](triangle<double> &tr) { return tr.print(std::cout); });
        } else if (comm == "exit") {
            break;
        } else if (comm == "map") {
            std::map<int, int, std::less<>, my_allocator<std::pair<const double, double>, 100000>> mp;
            for (int i = 0; i < 10; i++) {
                mp[i] = i;
            }
            std::for_each(mp.begin(), mp.end(), [](std::pair<double, double> X) {std::cout << X.first <<
" " << X.second << ", ";});
            std::cout << '\n';
        } else {
            std::cout << "ERROR: unknown command" << std::endl;
            continue;
        }
    }

    return 0;
}
```

allocator.h

```
#ifndef OOP6_ALLOCATOR_H
#define OOP6_ALLOCATOR_H


#include <cstdlib>
#include <cstdint>

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

#include "vector.h"


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():
                memory_pool_begin(new char[ALLOC_SIZE]),
                memory_pool_end(memory_pool_begin + ALLOC_SIZE),
                memory_pool_tail(memory_pool_begin)
        {};
```

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

    ~my_allocator() noexcept {
        delete[] memory_pool_begin;
    }

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

private:
    char* memory_pool_begin;
    char* memory_pool_end;
    char* memory_pool_tail;
    vector<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("This allocator can't allocate");
    }
    if(size_t(memory_pool_end - memory_pool_tail) < sizeof(T)){
        if(free_blocks.get_size()){
            auto it = free_blocks.begin();
            char* ptr = *it;
            free_blocks.erase(0);
            return reinterpret_cast<T*>(ptr);
        }
        throw std::bad_alloc();
    }
    T* result = reinterpret_cast<T*>(memory_pool_tail);
    memory_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("This allocator can't allocate");
    }
    if(ptr == nullptr){
        return;
    }
    free_blocks.push_back(reinterpret_cast<char*>(ptr));
}


#endif //OOP6_ALLOCATOR_H
```

vector.h

```
#ifndef OOP6_VECTOR_H
#define OOP6_VECTOR_H

#include <memory>
#include <iterator>
#include <utility>

template<class T> struct vector {

public:
    using value_type = T;
    using iterator = T *;

    iterator begin() const;

    iterator end() const;

    vector() : data(std::move(std::unique_ptr<T[]>(new T[1]))), size(0), cap(1) {};

    vector(int size) : data(std::move(std::unique_ptr<T[]>(new T[size]))), size(0), cap(size) {};

    T &operator[](int i);

    void push_back(const T &value);

    void erase(int pos);

    void resize(int NewSize);

    int get_size();

    ~vector() {};

private:
    std::unique_ptr<T[]> data;
    int size;
    int cap;
};

template<class T>
T &vector<T>::operator[](int i) {
    try{
        if (i >= size) {
            throw "ERROR: Vector size smaller";
        }
        return data[i];
    } catch(const char* f) {
        std::cout << f << "\n";
    }
}
```

```
template<class T>
void vector<T>::push_back(const T &value) {
    if (cap == size) {
        resize(size * 2);
    }
    data[size++] = value;
}

template<class T>
void vector<T>::erase(int pos) {
    std::unique_ptr<T[]> newd(new T[cap]);
    for (int i = 0; i < size; ++i) {
        if (i < pos) {
            newd[i] = data[i];
        } else if (i > pos) {
            newd[i - 1] = data[i];
        }
    }
    data = std::move(newd);
    size--;
}


template<class T>
void vector<T>::resize(int size) {
    std::unique_ptr<T[]> newd(new T[size]);
    int n = std::min(size, this->size);
    for (int i = 0; i < n; ++i) {
        newd[i] = data[i];
    }
    data = std::move(newd);
    this->size = n;
    cap = size;
}

template<class T>
int vector<T>::get_size() {
    return size;
}

template<class T>
typename vector<T>::iterator vector<T>::begin() const {
    return &data[0];
}

template<class T>
typename vector<T>::iterator vector<T>::end() const {
    return data[size];
}


#endif //OOP6_VECTOR_H
```

que.h

```
#ifndef OOP6_QUE_H
#define OOP6_QUE_H

#include <iterator>
#include <memory>
#include <utility>

template <class T, class Allocator = std::allocator<T>>
struct queue {
private:
    struct element;
public:
    queue() = default;

    // forw_it

    struct 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;

        forward_iterator(element *ptr);

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

        bool operator== (const forward_iterator& o) const;
        bool operator!= (const forward_iterator& o) const;
    private:
        element* ptr_ = nullptr;

        friend queue;
    };

    forward_iterator begin();
    forward_iterator end();

    void insert(forward_iterator& it, const T& value);
    void insert_to_num(int pos, const T& value);
    void erase(forward_iterator it);
    void erase_to_num(int pos);
    bool empty() {
        return first == nullptr;
```

```
    }

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

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>;


    // elem_que


    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 *endl = nullptr;
};


// it_func

template <class T, class Allocator>
typename queue<T, Allocator>::forward_iterator queue<T, Allocator>::begin() {
    if (first == nullptr) {
        return nullptr;
    }
    return forward_iterator(first.get());
}
```

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

// func_que

template <class T, class Allocator>
void queue<T, Allocator>::insert_to_num(int pos, const T& value) {
   forward_iterator iter = this -> begin();
   for (int i = 0; i < pos; ++i) {
      if (i == pos) {
         break;
      }
      ++iter;
   }
   this -> insert(iter, value);
}

template <class T, class Allocator>
void queue<T, Allocator>::insert(queue<T, Allocator>::forward_iterator& ptr, const T& value) {
    element* nd = this->allocator_.allocate(1);
    std::allocator_traits<allocator_type>::construct(this->allocator_, nd, value);
    auto val = unique_ptr(nd, deleter{&this->allocator_});
    forward_iterator it = this->begin();
    if (ptr == this->begin()) {
         val->next_element = std::move(first);
         first = std::move(val);
         return;
    }
    while ((it.ptr_ != nullptr) && (it.ptr_->next() != ptr)) {
         ++it;
    }
    if (it.ptr_ == nullptr) {
         throw std::logic_error ("ERROR");
    }
    val->next_element = std::move(it.ptr_->next_element);
    it.ptr_->next_element = std::move(val);
}

template <class T, class Allocator>
void queue<T, Allocator>::erase_to_num(int pos) {
   pos += 1;
   forward_iterator iter = this->begin();
   for (int i = 1; i <= pos; ++i) {
      if (i == pos) {
         break;
      }
      ++iter;
   }
   this->erase(iter);
```

```
}

template <class T, class Allocator>
void queue<T, Allocator>::erase(queue<T, Allocator>::forward_iterator ptr) {
    forward_iterator it = this->begin(), end = this->end();
    if (ptr == end) {
        throw std::logic_error("ERROR");
    }
    if (ptr == it) {
        this->pop();
        return;
    }
    while ((it.ptr_ != nullptr) && (it.ptr_->next() != ptr)) {
        ++it;
    }
    if (it.ptr_ == nullptr) {
        throw std::logic_error("ERROR");
    }
    it.ptr_->next_element = std::move(ptr.ptr_->next_element);
}

template <class T, class Allocator>
void queue<T, Allocator>::pop() {
    if (first == nullptr) {
        throw std::logic_error ("queue is empty");
    }
    auto tmp = std::move(first->next_element);
    first = std::move(tmp);
}

template <class T, class Allocator>
void queue<T, Allocator>::push(const T& value) {
    element* result = this->allocator_.allocate(1);
    std::allocator_traits<allocator_type>::construct(this -> allocator_, result, value);
    if (!first) {
        first = unique_ptr(result, deleter{&this->allocator_});
        endl = first.get();
        return;
    }
    endl->next_element = unique_ptr(result, deleter{&this->allocator_});
    endl = endl->next_element.get();
}

template <class T, class Allocator>
T& queue<T, Allocator>::top() {
    if (first == nullptr) {
        throw std::logic_error("queue is empty");
    }
    return first->value;
}

template<class T, class Allocator>
```

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

template<class T, class Allocator>
queue<T, Allocator>::forward_iterator::forward_iterator(queue<T, Allocator>::element *ptr) {
    ptr_ = ptr;
}

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

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

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

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

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



#endif //OOP6_QUE_H
```

Тесты

test_01.txt

```
add 0 0 0 1 1 0
add 0 0 0 2 2 0
print_all
pop
print_all
```

```
add 0 0 0 4 4 0
print_all
exit
```

res_01.txt

```
add 0 0 0 1 1 0
add 0 0 0 2 2 0
print_all
[0 0] [0 1] [1 0]
[0 0] [0 2] [2 0]
pop
print_all
[0 0] [0 2] [2 0]
add 0 0 0 4 4 0
print_all
[0 0] [0 2] [2 0]
[0 0] [0 4] [4 0]
exit
```

test_02.txt

```
insert 0 0 0 1 1 0
0
print_all
insert 0 0 0 2 2 0
0
print_all
insert 0 0 0 3 3 0
1
print_all
erase 0
print_all
erase 1
print_all
exit
```

res_02.txt

```
insert 0 0 0 1 1 0
0
print_all
[0 0] [0 1] [1 0]
insert 0 0 0 2 2 0
0
print_all
[0 0] [0 2] [2 0]
[0 0] [0 1] [1 0]
insert 0 0 0 3 3 0
1
print_all
[0 0] [0 2] [2 0]
```

[0 0] [0 3] [3 0]
[0 0] [0 1] [1 0]
erase 0
print_all
[0 0] [0 3] [3 0]
[0 0] [0 1] [1 0]
erase 1
print_all
[0 0] [0 3] [3 0]
exit

Ссылка на репозиторий на GitHub

https://github.com/AlexPshen/oop_exercise_06.git

Вывод

В данной лабораторной работе мы познакомились с аллокаторами. Аллокатор умеет выделять и освобождать память в требуемых количествах определённым образом. Также при правильном подходе можно значительно ускорить программу, нежели если бы мы использовали стандартную аллокацию памяти.