Московский Авиационный Институт
(Национальный исследовательский Университет)

Факультет: «Информационные технологии и прикладная математика»
Кафедра: 806 «Вычислительная математика и программирование»

Лабораторная работа
по курсу «ООП»

Тема:
Основы работы с коллекциями: итераторы.

Студент:	Симонов С.Я.
Группа:	М8О-206Б-18
Преподаватель:	Журавлев А.А.
Вариант:	21
Оценка:	
Дата:	

Москва
2019

1. Код на C++:

queue.hpp

```
#ifndef D_QUEUE_HPP_
#define D_QUEUE_HPP_ 1

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

namespace containers {

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

      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>;

    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;
};

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);
}

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(containers::queue<T, Allocator>::forward_iterator& ptr,
const T& value) {
    auto val = std::unique_ptr<element>(new element{value});
    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(containers::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(containers::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 // D_CONTAINERS_QUEUE_HPP_
```

vector.hpp

```
#ifndef MY_VECTOR_HPP_
#define MY_VECTOR_HPP_ 1

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

namespace containers {

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

        T& operator[] (int posision);
        void PushBack(const T& value);
        void erase(int pos);
        void resize(int NewSize);
        int GetSize();
        ~vector() {};

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

    template <class T>
    T& vector<T>::operator[] (int posision) {
        if (posision >= size) {
            throw std::logic_error("ERROR");
        }
        return data[posision];
    }

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

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


    template <class T>
    void vector<T>::resize(int size) {
```

```
        std::unique_ptr<T[]> newData(new T[size]);
        int n = std::min(size, this->size);
        for(int i = 0; i < n; ++i) {
            newData[i] = data[i];
        }
        data = std::move(newData);
        this->size = n;
        allocated = size;
    }

    template <class T>
    int vector<T>::GetSize() {
        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 //MY_VECTOR_HPP_
```

allocator.hpp

```
#ifndef D_MY_ALLOCATOR_H_
#define D_MY_ALLOCATOR_H_ 1

#include <cstdlib>
#include <cstdint>

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

#include "vector.hpp"


namespace all {
  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;
  containers::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 arrays");
  }
  if(size_t(memory_pool_end - memory_pool_tail) < sizeof(T)){
    if(free_blocks_.GetSize()){
      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 arrays");
  }
```

```
    if(ptr == nullptr){
      return;
    }
    free_blocks_.PushBack(reinterpret_cast<char*>(ptr));
  }
}

#endif // D_MY_ALLOCATOR_H_
```

main.cpp

```
#include <iostream>
#include <algorithm>
#include <map>

#include "rhombus.hpp"
#include "containers/queue.hpp"
#include "allocators/allocator.hpp"

int main() {
    int posision;
    containers::queue<rhombus<int>, all::my_allocator<rhombus<int>, 100>> q;

    std::map<int, int, std::less<>, all::my_allocator<std::pair<const int, int>, 1000>> mp;
            for (int i = 0; i < 8; i++) {
                mp[i] = i * i;
            }
            std::for_each(mp.begin(), mp.end(), [](std::pair<int, int> X) {std::cout << X.first << " "
<< X.second << ", ";});
            std::cout << std::endl;

    std::cout << "1 - push\n"
              << "2 - top\n"
              << "3 - pop\n"
              << "4 - erase_to_num\n"
              << "5 - for_each\n"
              << "6 - map\n"
              << "0 - exit\n";

    for (;;) {
        int command;
        std::cin >> command;
        if (command == 1) {
            rhombus<int> rhomb(std::cin);
            q.push(rhomb);
        } else if (command == 2) {
            q.top().print();
        } else if (command == 3) {
            q.pop();
        } else if (command == 4) {
            std::cin >> posision;
```

```
            q.erase_to_num(posision);
        } else if (command == 5) {
            std::for_each(q.begin(), q.end(), [](rhombus<int> &rhomb) { return rhomb.print(); });
        } else if (command == 0) {
            break;
        } else if (command == 6) {

            std::map<int, int, std::less<>, all::my_allocator<std::pair<const int, int>, 1000>> mp;
            for (int i = 0; i < 6; i++) {
                mp[i] = i * i;
            }
            std::for_each(mp.begin(), mp.end(), [](std::pair<int, int> X) {std::cout << X.first << " "
<< X.second << ", ";});
            std::cout << std::endl;
        } else {
            std::cout << "ERROR" << std::endl;
            continue;
        }
    }

    return 0;
}
```

rhombus.cpp

```
#ifndef D_RHOMBUS_HPP_
#define D_RHOMBUS_HPP_ 1

#include <algorithm>
#include <iostream>
#include <assert.h>
#include <cmath>

#include "vertex.hpp"

template<class T>
struct rhombus {
public:
    rhombus (std::istream& is);

    bool correct() const;

    vertex<double> center() const;
    double area() const;
    double perimeter() const;
    void print() const;
private:
    vertex<T> a1, a2, a3, a4;
};


template <class T>
```

```
rhombus<T>::rhombus(std::istream& is) {
   is >> a1 >> a2 >> a3 >> a4;
   assert(correct());
}

template <class T>
bool rhombus<T>::correct() const {
   T str1, str2, str3, str4;
   str1 = sqrt((a2.x - a1.x) * (a2.x - a1.x) + (a2.y - a1.y) * (a2.y - a1.y));
   str2 = sqrt((a3.x - a2.x) * (a3.x - a2.x) + (a3.y - a2.y) * (a3.y - a2.y));
   str3 = sqrt((a4.x - a3.x) * (a4.x - a3.x) + (a4.y - a3.y) * (a4.y - a3.y));
   str4 = sqrt((a1.x - a4.x) * (a1.x - a4.x) + (a1.y - a4.y) * (a1.y - a4.y));
   if (str1 == str2 && str2 == str3 && str3 == str4) {
      return true;
   }
   return false;
}

template <class T>
vertex<double> rhombus<T>::center() const {
   vertex<double> p;
   p.x = (a1.x + a2.x + a3.x + a4.x) / 4;
   p.y = (a1.y + a2.y + a3.y + a4.y) / 4;
   return p;
}

template <class T>
double rhombus<T>::area() const {
   const T s1 = 0.5 * abs((a2.x - a1.x) * (a3.y - a1.y) - (a3.x - a1.x) * (a2.y - a1.y));
   const T s2 = 0.5 * abs((a3.x - a1.x) * (a4.y - a1.y) - (a4.x - a1.x) * (a3.y - a1.y));
   return s1 + s2;
}

template <class T>
double rhombus<T>::perimeter() const {
   const T str1 = sqrt((a2.x - a1.x) * (a2.x - a1.x) + (a2.y - a1.y) * (a2.y - a1.y));
   const T str2 = sqrt((a3.x - a2.x) * (a3.x - a2.x) + (a3.y - a2.y) * (a3.y - a2.y));
   const T str3 = sqrt((a4.x - a3.x) * (a4.x - a3.x) + (a4.y - a3.y) * (a4.y - a3.y));
   const T str4 = sqrt((a1.x - a4.x) * (a1.x - a4.x) + (a1.y - a4.y) * (a1.y - a4.y));
   return str1 + str2 + str3 + str4;
}

template <class T>
void rhombus<T>::print() const {
   std::cout << a1 << ' ' << a2 << ' ' << a3 << ' ' << a4 << '\n';
}

#endif
```

vertex.hpp

```
#ifndef D_VERTEX_HPP_
```

```
#define D_VERTEX_HPP_ 1

#include <iostream>

template<class T>
struct vertex {
   T x;
   T y;
};

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, const vertex<T>& p) {
   os << '[' << ' ' << p.x << ' ' << p.y << ' ' << ']';
   return os;
}

template <class T>
vertex<T> operator+ (vertex<T> p1, vertex<T> p2) {
   vertex<T> p;
   p.x = p1.x + p2.x;
   p.y = p1.y + p2.y;
   return p;
}

template <class T>
vertex<T>& operator/ (vertex<T>& p, int num) {
   p.x = p.x / num;
   p.y = p.y / num;
   return p;
}

#endif // D_VERTEX_HPP_
```

2. Ссылка на репозиторий в GitHub:

https://github.com/keoni02032/oop_exercise_06

3. Набор testcases:

test_01.test

```
1
-1 0
0 1
```

1 0
0 -1
1
-2 0
0 2
2 0
0 -2
2
5
6
4
0
2
5
3
0

test_02.test
1
-1 0
0 1
1 0
0 -1
1
-2 0
0 2
2 0
0 -2
5
2
4
0
3
1
-2 0
0 2
2 0
0 -2
6
1
-3 0
0 3
3 0
0 -3
5
0

4.Результаты выполнения программы:

test_01.result

```
0 0, 1 1, 2 4, 3 9, 4 16, 5 25, 6 36, 7 49,
1 - push
2 - top
3 - pop
4 - erase_to_num
5 - for_each
6 - map
0 - exit
1
-1 0
0 1
1 0
0 -1
1
-2 0
0 2
2 0
0 -2
2
[ -1 0 ] [ 0 1 ] [ 1 0 ] [ 0 -1 ]
5
[ -1 0 ] [ 0 1 ] [ 1 0 ] [ 0 -1 ]
[ -2 0 ] [ 0 2 ] [ 2 0 ] [ 0 -2 ]
6
0 0, 1 1, 2 4, 3 9, 4 16, 5 25,
4
0
2
[ -2 0 ] [ 0 2 ] [ 2 0 ] [ 0 -2 ]
5
[ -2 0 ] [ 0 2 ] [ 2 0 ] [ 0 -2 ]
3
0
```

test_02.result

```
0 0, 1 1, 2 4, 3 9, 4 16, 5 25, 6 36, 7 49,
1 - push
2 - top
3 - pop
4 - erase_to_num
5 - for_each
6 - map
0 - exit
1
```

```
-1 0
0 1
1 0
0 -1
1
-2 0
0 2
2 0
0 -2
5
[ -1 0 ] [ 0 1 ] [ 1 0 ] [ 0 -1 ]
[ -2 0 ] [ 0 2 ] [ 2 0 ] [ 0 -2 ]
2
[ -1 0 ] [ 0 1 ] [ 1 0 ] [ 0 -1 ]
4
0
3
1
-2 0
0 2
2 0
0 -2
6
0 0, 1 1, 2 4, 3 9, 4 16, 5 25,
1
-3 0
0 3
3 0
0 -3
5
[ -2 0 ] [ 0 2 ] [ 2 0 ] [ 0 -2 ]
[ -3 0 ] [ 0 3 ] [ 3 0 ] [ 0 -3 ]
0
```

5. Объяснение результатов работы программы:

При запуске программы пользователю дается выбор из семи команд. При вводе «1» считываются координаты фигуры которые заносятся в очередь, память выделятся из аллкартора на динамическом массиве. При вводе «2» выводится первый элемент очереди, «3» удаляется первый элемент из очереди, «4» удаляется элемент по заданному номеру итератора, «5» выводятся все элементы хранящиеся в очереди, «6» демонстрируется совместимость работы с std::map, «0» происходит выход из программы.

6. Вывод:

В ходе выполнения данной лабораторной работы были получены навыки работы с аллокаторами. Аллокаторы позволяют ускорить

быстродействие работы программы, кроме того позволяют корректно выделять память под тот или иной элемент хранящийся в нашем контейнере.