Московский авиационный институт
(Национальный исследовательский университет)
Факультет "Информационные технологии и прикладная математика"

Лабораторная работа №6 по курсу “Объектно-ориентированное программирование”

Студент: Хисамутдинов Д.С.

Группа: М8О-208Б *Преподаватель:*

Журавлев А.А.

Вариант: 5

Оценка: Дата:

Москва
2019

1 Исходный код

vertex.hpp

```
#pragma once

#include        <iostream>
#include        <cmath>
#include        <iomanip>

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

        template<class T>
        std::istream& operator>>(std::istream& is, vertex_t<T>& p) {
        is >> p.x >> p.y;
        return is;
        }

template<class T>
std::ostream& operator<<(std::ostream& os, const vertex_t<T>& p) { os << std::fixed <<
std::setprecision(3) << "[" << p.x << ",
      " << p.y << "]";
        return os;
        }

        template<class T>
        T calculateDistance(const vertex_t<T>& p1, const vertex_t<T>& p2)
       {
        return sqrt(pow(p2.x - p1.x, 2) + pow(p2.y - p1.y, 2));
        }

                        template<class T>
                        T triangleArea(vertex_t<T> p1, vertex_t<T> p2, vertex_t<T> p3) {
                        return 0.5 * fabs((p1.x - p3.x) * (p2.y - p3.y) - (p2.x - p3.x ) * (p1.y
                        - p3.y));
                        }
```

rhombus.hpp

```
#pragma once

#include <array>

#include "vertex.hpp"

                template<class T>
                double checkIfRhombus(const vertex_t<T> p1, const vertex_t<T>& p2,
                const vertex_t<T>& p3, const vertex_t<T>& p4) {
                T d1 = calculateDistance(p1, p2);
```

```
11              T d2 = calculateDistance(p1, p3);
12              T d3 = calculateDistance(p1, p4);
13              if(d1 == d2) {
14              return d3;
15              } else if(d1 == d3) {
16              return d2;
17              } else if(d2 == d3) {
18              return d1;
19              } else {
20              throw std::invalid_argument("Entered coordinates are not forming Rhombus. Try
                entering new coordinates");
21              }
22              }
23
24  template <class T>
25  struct Rhombus {
26  std::array<vertex_t<T>, 4> points; 27      T smallerDiagonal, biggerDiagonal;
28                      Rhombus(const vertex_t<T>& p1, const vertex_t<T>& p2, const vertex_t<T>& p3,
29                      const vertex_t<T>& p4);
30                      double area() const;
31                      vertex_t<T> center() const;
32                      void print(std::ostream& os) const;
33                      };
34
35                              template<class T>
36                              Rhombus<T>::Rhombus(const vertex_t<T>& p1, const vertex_t<T>& p2,
37                              const vertex_t<T>& p3, const vertex_t<T>& p4) {
38                              try {
39                              T d1 = checkIfRhombus(p1, p2, p3, p4);
40                              T d2 = checkIfRhombus(p2, p1, p3, p4);
41                              T d3 = checkIfRhombus(p3, p1, p2, p4);
42                              T d4 = checkIfRhombus(p4, p1, p2, p3);
43                              if(d1 == d2 || d1 == d4) {
44                              if(d1 < d3) {
45                              smallerDiagonal = d1;
46                              biggerDiagonal = d3;
47                              } else {
48                              smallerDiagonal = d3;
49                              biggerDiagonal = d1;
50                              }
51                              } else if(d1 == d3) {
52                              if(d1 < d2) {
53                              smallerDiagonal = d1;
54                              biggerDiagonal = d2;
55                              } else {
56                              smallerDiagonal = d2;
57                              biggerDiagonal = d1;
58                              }
59                              }
60                              } catch(std::exception& e) {
61                              throw std::invalid_argument(e.what());
62                              return;
```

```
                        }
                        points[0] = p1;
                        points[1] = p2;
                        points[2] = p3;
                        points[3] = p4;
                        }

template<class T>
double Rhombus<T>::area() const {
return smallerDiagonal * biggerDiagonal / 2.0;
}

        template<class T>
        vertex_t<T> Rhombus<T>::center() const {
        if(calculateDistance(points[0], points[1]) == smallerDiagonal
||
        calculateDistance(points[0], points[1]) == biggerDiagonal) {
        return {((points[0].x + points[1].x) / 2.0), ((points[0].y
+ points[1].y) / 2.0)};
        } else if(calculateDistance(points[0], points[2]) == smallerDiagonal ||
        calculateDistance(points[0], points[2]) == biggerDiagonal) {
        return {((points[0].x + points[2].x) / 2.0), ((points[0].y
+ points[2].y) / 2.0)};
        } else {
        return {((points[0].x + points[3].x) / 2.0), ((points[0].y
+ points[3].y) / 2.0)};
        }
        }

        template<class T>
        void Rhombus<T>::print(std::ostream& os) const {
        os << "Rhombus: ";
        for(const auto& p : points) {
        os << p << ' ';
        }
        os << std::endl;
}
```

stack.hpp

```
#pragma once

 #include  <iterator>  #include
<memory>
#include <iostream>

namespace cntrs {

template<class T, class Allocator = std::allocator<T>> class stack_t { private:
        struct node_t;
        public:
        struct forward_iterator {
                using      value_type = T;
```

```
16          using     reference = T&;
17          using     pointer = T*;
18          using     difference_type = ptrdiff_t;
19          using     iterator_category = std::forward_iterator_tag;
20                  forward_iterator(node_t* ptr) : ptr_(ptr) {};
21                  T& operator*();
22                  forward_iterator& operator++();
23                  forward_iterator operator++(int);
24                  bool operator==(const forward_iterator& it) const;
25                  bool operator!=(const forward_iterator& it) const;
26                  private:
27                  node_t* ptr_;
28                  friend stack_t;
29                  };
30
31    forward_iterator begin();
32    forward_iterator end();
33    void     insert(const forward_iterator& it, const                    T& value);
34    void     insert(const int& pos, const T& value);
35    void     erase(const forward_iterator& it);
36    void     erase(int pos);
37    void     pop();
38    T top();
39    void push(const T& value);
40    stack_t() = default;
41    stack_t(const stack_t&) = delete; 42 private:
43  using allocator_type = typename Allocator::template rebind< node_t>::other;
44
45          struct deleter {
46          deleter(allocator_type* allocator) : allocator(allocator)
    {};
47
48                      void operator()(node_t* ptr) {
49                      if(ptr != nullptr) {
50                      std::allocator_traits<allocator_type>::destroy(* allocator, ptr);
51                      allocator->deallocate(ptr, 1);
52                      }
53                      }
54
55                  private:
56                  allocator_type* allocator;
57                  };
58
59          struct node_t {
60          T value;
61          std::unique_ptr<node_t, deleter> nextNode{nullptr, deleter {&this->allocator}};
62          forward_iterator next();
63          node_t(const T& value, std::unique_ptr<node_t, deleter> next) : value(value),
          nextNode(std::move(next)) {};
64          };
65          std::unique_ptr<node_t, deleter> head{nullptr, deleter{&this-> allocator}};
66          node_t* tail = nullptr;
67          stack_t& operator=(const stack_t&);
```

```
68                  allocator_type allocator {};
69                  };
70
71          template<class T, class Allocator>
72          typename stack_t<T, Allocator>::forward_iterator stack_t<T,
          Allocator>::node_t::next() {
73          return nextNode.get();
74          }
75
76  template<class T, class Allocator>
77  T& stack_t<T, Allocator>::forward_iterator::operator*() { 78          return ptr_->value;
79 }
80
81          template<class T, class Allocator>
82          typename stack_t<T, Allocator>::forward_iterator& stack_t<T,
          Allocator>::forward_iterator::operator++() {
83          *this = ptr_->next();
84          return *this;
85          }
86
87          template<class T, class Allocator>
88          typename stack_t<T, Allocator>::forward_iterator stack_t<T,
          Allocator>::forward_iterator::operator++(int) {
89          forward_iterator old = *this;
90          ++*this;
91          return old;
92          }
93
94          template<class T, class Allocator>
95          bool stack_t<T, Allocator>::forward_iterator::operator!=(const forward_iterator& it) const {
96          return ptr_ != it.ptr_;
97          }
98
99           template<class T, class Allocator>
100          bool stack_t<T, Allocator>::forward_iterator::operator==(const forward_iterator& it) const {
101          return ptr_ == it.ptr_;
102          }
103
104          template<class T, class Allocator>
105          typename stack_t<T, Allocator>::forward_iterator stack_t<T,
          Allocator>::begin() {
106          return head.get();
107          }
108
109          template<class T, class Allocator>
110          typename stack_t<T, Allocator>::forward_iterator stack_t<T,
          Allocator>::end() {
111          return nullptr;
112          }
113
114                      template<class T, class Allocator>
115                      void stack_t<T, Allocator>::insert(const forward_iterator& it, const T& value) {
```

```
116                 node_t* ptr = this->allocator.allocate(1);
117                 std::allocator_traits<allocator_type>::construct(this-> allocator, ptr, value,
                    std::unique_ptr<node_t,
118                 deleter>(nullptr, deleter{&this->allocator}));
119                 std::unique_ptr<node_t, deleter> newNode(ptr, deleter{&this-> allocator});
120                 if(head == nullptr) {
121                 head = std::move(newNode);
122                 } else if(head->nextNode == nullptr) {
123                 if(it.ptr_) {
124                 tail = head.get();
125                 newNode->nextNode = std::move(head);
126                 head = std::move(newNode);
127                 } else {
128                 tail = newNode.get();
129                 head->nextNode = std::move(newNode);
130                 }
131                 } else if(head.get() == it.ptr_) {
132                 newNode->nextNode = std::move(head);
133                 head = std::move(newNode);
134                 } else if(it.ptr_ == nullptr) {
135                 tail->nextNode = std::move(newNode);
136                 tail = newNode.get();
137                 } else {
138                 auto temp = this->begin();
139                 while(temp.ptr_->next() != it.ptr_) {
140                 ++temp;
141                 }
142
143             newNode->nextNode = std::move(temp.ptr_->nextNode);
144             temp.ptr_->nextNode = std::move(newNode);
145             }
146             }
147
148                 template<class T, class Allocator>
149                 void stack_t<T, Allocator>::insert(const int& pos, const T& value)
    {
150                 int i = 0;
151                 auto temp = this->begin();
152                 if(pos == 0) {
153                 insert(temp, value);
154                 return;
155                 }
156                 while(i < pos) {
157                 if(temp.ptr_ == nullptr) {
158                 break;
159                 }
160                 ++temp;
161                 ++i;
162                 }
163                 if(i < pos) {
164                 throw std::logic_error("Out of bounds");
165                 }
```

```
166                     this->insert(temp, value);
167                     }
168
169                     template<class T, class Allocator>
170                     void stack_t<T, Allocator>::erase(const forward_iterator& it) {
171                     if(it == nullptr) {
172                     throw std::logic_error("Invalid iterator");
173                     }
174                     if(head == nullptr) {
175                     throw std::logic_error("Deleting from empty list");
176                     }
177                     if(it == this->begin()) {
178                     head = std::move(head->nextNode);
179                     } else {
180                     auto temp = this->begin();
181                     while(temp.ptr_->next() != it.ptr_) {
182                     ++temp;
183                     }
184                     temp.ptr_->nextNode = std::move(it.ptr_->nextNode);
185                     }
186                     }
187
188 template<class T, class Allocator>
189 void      stack_t<T, Allocator>::erase(int                    pos) {
190         auto temp = this->begin();
191         int i = 0;
192         while(i < pos) {
193                 if(temp.ptr_ == nullptr) {
194                         break;
195                 }
196                 ++temp;
197                 ++i;
198         }
199         if(temp.ptr_ == nullptr) {
200                 throw std::logic_error("Out                 of    bounds");
201         }
202         erase(temp);
203 }
204
205 template<class T, class Allocator>
206         void stack_t<T, Allocator>::pop() {
207         erase(this->begin());
208         }
209
210                 template<class T, class Allocator>
211                 T stack_t<T, Allocator>::top() {
212                 if(head) {
213                 return head->value;
214                 } else {
215                 throw std::logic_error("Stack                 is    empty");
216                 }
217                 }
218
219 template<class T, class Allocator>
```

```
220 void stack_t<T, Allocator>::push(const 221                    T& value) {
        insert(this->begin(), value);
222 }
223
224 }
```

allocator.hpp

```
1 #pragma once
2
3  #include <iostream>
4  #include <type_traits>
5
6  #include "tvector.hpp"
7  #include "stack.hpp"
8
9 namespace allctr {
10
11          template<class T, size_t ALLOC_SIZE>
12          struct allocator_t {
13          using value_type = T;
14          using size_type = size_t;
15          using difference_type = std::ptrdiff_t;
16          using is_always_equal = std::false_type;
17
18                  template<class U>
19                  struct rebind {
20                  using other = allocator_t<U, ALLOC_SIZE>;
21                  };
22
23                  allocator_t()  :  memory_pool_begin(new  char[ALLOC_SIZE]),
                    memory_pool_end(memory_pool_begin + ALLOC_SIZE),
24                  memory_pool_tail(memory_pool_begin) {};
25
26          allocator_t(const allocator_t&) = delete;
27          allocator_t(allocator_t&&) = delete;
28
29
30                  ~allocator_t() {
31                  delete[] memory_pool_begin;
32                  }
33                  T* allocate(size_t n); 34       void deallocate(T* ptr, size_t n);
35          private:
36          char* memory_pool_begin;
37          char* memory_pool_end;
38          char* memory_pool_tail;
39          cntrs::vector_t<char*> free_blocks;
40          };
41
42                          template<class T, size_t ALLOC_SIZE>
43                          T* allocator_t<T, ALLOC_SIZE>::allocate(size_t n) {
44                          if(n != 1) {
45                          throw std::logic_error("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(it);
                    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 allocator_t<T, ALLOC_SIZE>::deallocate(T* ptr, size_t n) {
                if(n != 1) {
                throw std::logic_error("Can't allocate arrays");
                }
                if(ptr == nullptr) {
                return;
                }
                free_blocks.push_back(reinterpret_cast<char*>(ptr));
                }
                }
```

tvector.hpp

```
#pragma once

#include <memory>

const int GROWTH = 2;

namespace cntrs {

 template <class T>
class vector_t { public:
        using value_type = T;
        using iterator = T*;
        vector_t() : data(std::move(std::unique_ptr<T[]>(new T[GROWTH
       ]))), size(0), allocated(GROWTH) {};
        vector_t(size_t size) : data(std::move(std::unique_ptr<T[]>( new T[size]))), size(0), allocated(size)
        {};
        void push_back(const T& item);
        void resize(size_t size);
        void erase(iterator pos);
        size_t getSize() const;
        T& operator[](size_t pos);
        iterator begin() const;
        iterator end() const;
        ~vector_t() {};
```

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

        template <class T>
        void vector_t<T>::push_back(const T& item) {
        if(size == allocated) {
        this->resize(size * GROWTH);
        }
        data[size++] = item;
        }

        template <class T>
        void vector_t<T>::resize(size_t 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>
        void vector_t<T>::erase(typename vector_t<T>::iterator pos) {
        auto end = this->end();
        while(pos != end) {
        *pos = *(pos + 1);
        ++pos;
        }
        this->size--;
        }

template<class T>
size_t vector_t<T>::getSize() const {
return size;
}

        template<class T>
        T& vector_t<T>::operator[](size_t pos) {
        if(pos >= this->size) {
        throw std::out_of_range("out of range");
        }
        return data[pos];
        }

template<class T>
typename vector_t<T>::iterator vector_t<T>::begin() const {
return data.get();
```

```
76          }
77
78                  template<class T>
79                  typename vector_t<T>::iterator vector_t<T>::end() const {
80                  if(data) {
81                  return data.get() + size;
82                  }
83
84          return nullptr;
85          }
86
87
88  }
```

main.cpp

```
1  #include        <iostream>
2  #include        <algorithm>
3  #include        <map>
4
5  #include        "stack.hpp"
6  #include        "rhombus.hpp"
7  #include        "allocator.hpp"
8
9                                   int main() {
10                                  std::map<int, int, std::less<int>,
11                                  allctr::allocator_t<std::pair<const int, int>, 1000>> m;
12                                  for(int i = 0; i < 10; ++i) {
13                                  m[i] = i;
14                                  }
15                                  m.erase(1);
16                                  cntrs::stack_t<Rhombus<double>,          allctr::allocator_t<Rhombus<
                                    double>, 1000>> s;
17                                  int command, pos;
18                                  std::cout << "1 - add element to stack(push/insert by iterator )" <<
                                    std::endl;
19                                  std::cout << "2 - delete element from stack(pop/erase by index
        /erase by iterator)" << std::endl;
20                                  std::cout << "3 - range-based for print" << std::endl;
21                                  std::cout << "4 - count_if example" << std::endl;
22                                  std::cout << "5 - top element" << std::endl;
23                                  std::cin >> command;
24                                  while(true) {
25                                  if(command == 0) {
26                                  break;
27                                  } else if(command == 1) {
28                                  std::cout << "Enter coordinates" << std::endl;
29                                  vertex_t<double> v1, v2, v3, v4;
30                                  std::cin >> v1 >> v2 >> v3 >> v4;
31                                  try {
32                                  Rhombus<double> r{v1, v2, v3, v4};
33                                  } catch(std::exception& e) {
34                                  std::cout << e.what() << std::endl;
```

```
std::cin >> command;
continue;
}
Rhombus<double> r{v1, v2, v3, v4};
std::cout << "1 - push to stack" << std::endl;
std::cout << "2 - insert by iterator" << std::endl;
std::cin >> command;
if(command == 1) {
s.push(r);
} else if(command == 2) {
std::cout << "Enter index" << std::endl;
std::cin >> pos;
s.insert(pos, r);
} else {
std::cout << "Wrong command" << std::endl;
std::cin >> command;
continue;
}
} else if(command == 2) {
std::cout << "1 - pop" << std::endl;
std::cout << "2 - erase by index" << std::endl;
std::cout << "3 - erase by iterator" << std::endl;
std::cin >> command;
if(command == 1) {
s.pop();
} else if(command == 2) {
std::cout << "Enter index" << std::endl;
std::cin >> pos;
s.erase(pos);
} else if(command == 3) {
std::cout << "Enter index" << std::endl;
std::cin >> pos;
auto temp = s.begin();
for(int i = 0; i < pos; ++i) {
++temp;
}
s.erase(temp);
} else {
std::cout << "Wrong command" << std::endl;
std::cin >> command;
continue;
}
} else if(command == 3) {
for(const auto& item : s) {
item.print(std::cout);
}
} else if(command == 4) {
std::cout << "Enter required square" << std::endl;
std::cin >> pos;
std::cout << "Number of rhombes with area less than " << pos << " equals
";
```

```
85                          std::cout << std::count_if(s.begin(), s.end(), [pos]( Rhombus<double> r)
                            {return r.area() < pos;}) << std::endl;
86                          } else if(command == 5) {
87                          try {
88                          s.top();
89                          } catch(std::exception& e) {
90                          std::cout << e.what() << std::endl;
91                          std::cin >> command;
92                          continue;
93                          }
94                          Rhombus<double> temp = s.top();
95                          std::cout << "Top: ";
96                          temp.print(std::cout);
97                          } else {
98                          std::cout << "Wrong command" << std::endl;
99                          }
100                         std::cin >> command;
101                         }
102                         return 0;
103                         }
```

CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.1)
2
3 project(lab6)
4
5         add_executable(lab6
6         main.cpp)
7
8 set_property(TARGET lab6 PROPERTY CXX_STANDARD 17)
9
10 set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -g -Wall -Wextra -Werror")
```

2 Тестирование

Набор входных данных для всех тестов одинаковый - ромбы с координатами ([-1, -1], [-1, 1], [1, 1], [1, -1]), ([-2, -2], [-2, 2], [2, 2], [2, -2]), ([-3, -3], [-3, 3], [3, 3], [3, -3]), ([-4, -4], [-4, 4], [4, 4], [4, -4]). Различия заключаются в методах добавления и удаления этих фигур в стек.

test_01.txt:

Добавим фигуры в стек с помощью метода push и напечатаем их. Затем с помощью count_if найдем количество ромбов с площадями меньше 4, 16, 36, 64, 81(0, 1, 2, 3, 4 соответственно). Удалим все фигуры из стека с помощью метода pop, перед каждым вызовом которого, выведем элемент на верху стека с помощью функции top.

Результат:

1 - add element to stack(push/insert by iterator)

2 - delete element from stack(pop/erase by index/erase by iterator)

3 - range-based for print

```
4 - count_if example
5 - top element
Enter coordinates
1 - push to stack
2 - insert by iterator
Enter coordinates
1 - push to stack
2 - insert by iterator
Enter coordinates
1 - push to stack
2 - insert by iterator
Enter coordinates
1 - push to stack
2 - insert by iterator
Rhombus: [-4.000, -4.000] [-4.000, 4.000] [4.000, 4.000] [4.000, -4.000]
Rhombus: [-3.000, -3.000] [-3.000, 3.000] [3.000, 3.000] [3.000, -3.000]
Rhombus: [-2.000, -2.000] [-2.000, 2.000] [2.000, 2.000] [2.000, -2.000]
Rhombus: [-1.000, -1.000] [-1.000, 1.000] [1.000, 1.000] [1.000, -1.000]
Enter required square
Number of rhombes with area less than 4 equals 0
Enter required square
Number of rhombes with area less than 16 equals 1 Enter
required square
Number of rhombes with area less than 36 equals 3 Enter
required square
Number of rhombes with area less than 64 equals 3 Enter
required square
Number of rhombes with area less than 81 equals 4
Top: Rhombus: [-4.000, -4.000] [-4.000, 4.000] [4.000, 4.000] [4.000, -4.000]
1 - pop
2 - erase by index
3 - erase by iterator
Top: Rhombus: [-3.000, -3.000] [-3.000, 3.000] [3.000, 3.000] [3.000, -3.000]
1 - pop
2 - erase by index
3 - erase by iterator
Top: Rhombus: [-2.000, -2.000] [-2.000, 2.000] [2.000, 2.000] [2.000, -2.000]
1 - pop
2 - erase by index
3 - erase by iterator
Top: Rhombus: [-1.000, -1.000] [-1.000, 1.000] [1.000, 1.000] [1.000, -1.000]
1 - pop
2 - erase by index
3 - erase by iterator Stack is empty
```

test_02.txt

То же самое, что и предыдущем тесте, кроме того, что фигуры добавляются в стек по итератору на 0,1,1,2 места соответственно.

Результат:

```
1 - add element to stack(push/insert by iterator)
2 - delete element from stack(pop/erase by index/erase by iterator)
3 - range-based for print
4 - count_if example
5 - top element
Enter coordinates
1 - push to stack
2 - insert by iterator
Enter index
Enter coordinates
1 - push to stack
2 - insert by iterator
Enter index
Enter coordinates
1 - push to stack
2 - insert by iterator
Enter index
Enter coordinates
1 - push to stack
2 - insert by iterator
Enter index
Rhombus: [-1.000, -1.000] [-1.000, 1.000] [1.000, 1.000] [1.000, -1.000]
Rhombus: [-3.000, -3.000] [-3.000, 3.000] [3.000, 3.000] [3.000, -3.000]
Rhombus: [-4.000, -4.000] [-4.000, 4.000] [4.000, 4.000] [4.000, -4.000]
Rhombus: [-2.000, -2.000] [-2.000, 2.000] [2.000, 2.000] [2.000, -2.000]
Enter required square
Number of rhombes with area less than 4 equals 0
Enter required square
Number of rhombes with area less than 16 equals 1 Enter
required square
Number of rhombes with area less than 36 equals 3
Enter required square
Number of rhombes with area less than 64 equals 3 Enter
required square
Number of rhombes with area less than 81 equals 4
Top: Rhombus: [-1.000, -1.000] [-1.000, 1.000] [1.000, 1.000] [1.000, -1.000]
1 - pop
2 - erase by index
3 - erase by iterator
Top: Rhombus: [-3.000, -3.000] [-3.000, 3.000] [3.000, 3.000] [3.000, -3.000]
1 - pop
```

2 - erase by index
3 - erase by iterator
Top: Rhombus: [-4.000, -4.000] [-4.000, 4.000] [4.000, 4.000] [4.000, -4.000]
1 - pop
2 - erase by index
3 - erase by iterator
Top: Rhombus: [-2.000, -2.000] [-2.000, 2.000] [2.000, 2.000] [2.000, -2.000]
1 - pop
2 - erase by index
3 - erase by iterator Stack is empty

test_03.txt

То же самое, что и предыдущем тесте, кроме того, что фигуры удаляются из стека по индексу в следующем порядке: 3-я, 3-я, 1-я, 1-я. После каждого удаления происходит печать стека.

Результат:

1 - add element to stack(push/insert by iterator)
2 - delete element from stack(pop/erase by index/erase by iterator)
3 - range-based for print
4 - count_if example
5 - top element
Enter coordinates
1 - push to stack
2 - insert by iterator
Enter index
Enter coordinates
1 - push to stack
2 - insert by iterator
Enter index
Enter coordinates
1 - push to stack
2 - insert by iterator
Enter index
Enter coordinates
1 - push to stack
2 - insert by iterator
Enter index
Rhombus: [-1.000, -1.000] [-1.000, 1.000] [1.000, 1.000] [1.000, -1.000]
Rhombus: [-3.000, -3.000] [-3.000, 3.000] [3.000, 3.000] [3.000, -3.000]
Rhombus: [-4.000, -4.000] [-4.000, 4.000] [4.000, 4.000] [4.000, -4.000]
Rhombus: [-2.000, -2.000] [-2.000, 2.000] [2.000, 2.000] [2.000, -2.000]
Enter required square
Number of rhombes with area less than 4 equals 0
Enter required square

```
Number of rhombes with area less than 16 equals 1 Enter
required square
Number of rhombes with area less than 36 equals 3 Enter
required square
Number of rhombes with area less than 64 equals 3 Enter
required square
Number of rhombes with area less than 81 equals 4
Top: Rhombus: [-1.000, -1.000] [-1.000, 1.000] [1.000, 1.000] [1.000, -1.000]
1 - pop
2 - erase by index
3 - erase by iterator
Enter index
Rhombus: [-1.000, -1.000] [-1.000, 1.000] [1.000, 1.000] [1.000, -1.000]
Rhombus: [-3.000, -3.000] [-3.000, 3.000] [3.000, 3.000] [3.000, -3.000]
Rhombus: [-2.000, -2.000] [-2.000, 2.000] [2.000, 2.000] [2.000, -2.000]
1 - pop
2 - erase by index
3 - erase by iterator
Enter index
Rhombus: [-1.000, -1.000] [-1.000, 1.000] [1.000, 1.000] [1.000, -1.000]
Rhombus: [-3.000, -3.000] [-3.000, 3.000] [3.000, 3.000] [3.000, -3.000]
1 - pop
2 - erase by index
3 - erase by iterator
Enter index
Rhombus: [-3.000, -3.000] [-3.000, 3.000] [3.000, 3.000] [3.000, -3.000]
1 - pop
2 - erase by index
3 - erase by iterator
Enter index
```

3 Объяснение результатов работы программы

При вводе координат для создания ромба производится проверка этих координат, ведь они могут не образовывать ромб. Для этого реализована функция checkIfRhombus, которая вычисляет расстояния от одной точки до трёх остальных, а поскольку фигура является ромбом, то два из низ должны быть равны. Третье же значение функция возвращает ведь оно равно длине одной из диагоналей. Площадь ромба вычисляется как половина произведения диагоналей, центр - точка пересечения диагоналей.

4 Выводы

Умные указатели при грамотном использовании позволяют сильно сэкономить время на выявление утечек памяти и исправления их. Однако при первом их использовании не так просто написать корректно работающую программу, ведь они несколько отличаются от сырых указателей и, соответственно, методов работы с ними.