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: «Основные работы с коллекциями: итераторы»

Группа:	М8О-206Б-18, №12
Студент:	Кузьмичев Александр Николаевич
Преподаватель:	Журавлёв Андрей Андреевич
Оценка:	
Дата:	22.12.2019

Москва, 2019

1. Задание

Разработать программу на языке C++ согласно варианту задания. Программа на C++ должна собираться с помощью системы сборки CMake. Программа должна получать данные из стандартного ввода и выводить данные в стандартный вывод.

Необходимо настроить сборку лабораторной работы с помощью CMake. Собранная программа должна называться oop_exercise_06 (в случае использования Windows oop_exercise_06.exe)

Необходимо зарегистрироваться на GitHub (если студент уже имеет регистрацию на GitHub то можно использовать ее) и создать репозитарий для задания лабораторной работы.

Преподавателю необходимо предъявить ссылку на публичный репозиторий на Github. Имя репозитория должно быть https://github.com/login/oop_exercise_06

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:

Стек – pop, push, top;

Очередь – pop, push, top;

Список,Динамический массив – доступ к элементу по оператору [];

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);

Коллекция должна использовать аллокатор для выделеления и освобождения памяти для своих элементов.

Аллокатор должен быть совместим с контейнерами std::map и std::list (опционально – vector).

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;

2. Адрес репозитория на GitHub

https://github.com/poisoned-monkey/oop_exercise_06

3. Код программы на C++

main.cpp

```
#include<iostream>
#include<algorithm>
#include<locale.h>
#include"trapeze.h"
#include"containers.h"
#include"allocators.h"

void Menu1() {
   std::cout << "1. Добавить фигуру в список\n";
   std::cout << "2. Удалить фигуру\n";
   std::cout << "3. Вывести фигуру\n";
   std::cout << "4. Вывести все фигуры\n";
   std::cout << "5. Вывести кол-во фигур чья площаль больше чем ...\n";
}

void Push() {
   std::cout << "1. Добавить фигуру в начало списка\n";
   std::cout << "2. Добавить фигуру в конец списка\n";
   std::cout << "3. Добавить фигуру по индексу\n";
}

void Delete() {
   std::cout << "1. Удалить фигуру в начале списка\n";
   std::cout << "2. Удалить фигуру в конце списка\n";
   std::cout << "3. Удалить фигуру по индексу\n";
}

void Print() {
   std::cout << "1. Вывести первую фигуру в списке\n";
   std::cout << "2. Вывести последнюю фигуру в списке\n";
   std::cout << "3. Вывести фигуру по индексу\n";
}

int main() {
   std::cout<<sizeof(Trapeze<int>)<<std::endl;
   setlocale(LC_ALL, "rus");
   containers::list<Trapeze<int>, allocators::my_allocator<Trapeze<int>,
500>> MyList;
   Trapeze<int> TempTrapeze;
   while (true) {
      Menu1();
```

```
      int n, m, ind;
      double s;
      std::cin >> n;
      switch (n) {
      case 1:
         TempTrapeze.read(std::cin);
         Push();
         std::cin >> m;
         switch (m) {
         case 1:
            MyList.push_front(TempTrapeze);
            break;
         case 2:
            MyList.push_back(TempTrapeze);
            break;
         case 3:
            std::cin >> ind;
            MyList.insert_by_number(ind, TempTrapeze);
         default:
            break;
         }
         break;
      case 2:
         Delete();
         std::cin >> m;
         switch (m) {
         case 1:
            MyList.pop_front();
            break;
         case 2:
            MyList.pop_back();
            break;
         case 3:
            std::cin >> ind;
            MyList.delete_by_number(ind);
            break;
         default:
            break;
         }
         break;
      case 3:
         Print();
         std::cin >> m;
         switch (m) {
         case 1:
            MyList.front().print(std::cout);
            std::cout << std::endl;
            break;
         case 2:
            MyList.back().print(std::cout);
            std::cout  << std::endl;
            break;
         case 3:
            std::cin >> ind;
            MyList[ind].print(std::cout);
            std::cout  << std::endl;
            break;
         default:
            break;
         }
         break;
      case 4:
         std::for_each(MyList.begin(), MyList.end(), [](Trapeze<int> &X)
{ X.print(std::cout); std::cout << std::endl; });
         break;
      case 5:
```

```
            std::cin >> s;
            std::cout << std::count_if(MyList.begin(), MyList.end(), [=]
(Trapeze<int>& X) {return X.square() > s; }) << std::endl;
            break;
        default:
            return 0;
        }
    }
    system("pause");
    return 0;
}
```

vertex.h

```
#pragma once
#include<iostream>
#include<cmath>
template<class T>
class Vertex {
public:
    T x, y;
};

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

template<class T>
std::ostream& operator<<(std::ostream& os, Vertex<T> point) {
    os <<'['<< point.x << ",  " << point.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> operator+=(Vertex<T> &a, const Vertex<T> &b) {
    a.x += b.x;
    a.y += b.y;
    return a;
}

template<class T>
double distance(const Vertex<T> &a, const Vertex<T>& b) {
    return sqrt(pow(a.x - b.x, 2) + pow(a.y - b.y, 2));
}
```

trapeze.h

```
#pragma once
#include"vertex.h"
template <class T>
class Trapeze {
private:
    Vertex<T> Vertexs[4];
```

```
public:
    using vertex_type = Vertex<T>;
    Trapeze();
    Trapeze(std::istream& in);
    void read(std::istream& in);
    Vertex<T> center() const;
    double square() const;
    void print(std::ostream& os) const;
    friend std::ostream& operator<< (std::ostream &out, const Trapeze<T>
&point);
    friend std::ostream& operator>> (std::istream &in, const Trapeze<T>
&point);
};

template<class T> Trapeze<T>::Trapeze() {}

template<class T> Trapeze<T>::Trapeze(std::istream& in) {
    for (int i = 0; i < 4; i++)
        in >> Vertexs[i];
}
template<class T> double Trapeze<T>::square() const {
    double Area = 0;
    for (int i = 0; i < 4; i++) {
        Area += (Vertexs[i].x) * (Vertexs[(i + 1) % 4].y) - (Vertexs[(i + 1)
% 4].x)*(Vertexs[i].y);
    }
    Area *= 0.5;
    return abs(Area);
}

template<class T> void Trapeze<T>::print(std::ostream& os) const {
    os << "Trapeze: ";
    for (int i = 0; i < 4; i++)
        os << Vertexs[i] << ' ';
    os << '\n';
}


template<class T> Vertex<T> Trapeze<T>::center() const {
    Vertex<T> res = Vertex<T>();
    for (int i = 0; i < 4; i++)
        res += Vertexs[i];
    return res / 4;
}

template <class T> void Trapeze<T>::read(std::istream& in) {
    Trapeze<T> res = Trapeze(in);
    *this = res;
}

template<class T>
std::ostream& operator<< (std::ostream &out, const Trapeze<T> &point) {
    out << "Trapeze: ";
    for (int i = 0; i < 4; i++)
        out << point.Vertexs[i] << ' ';
    out << '\n';
}

template<class T>
std::ostream& operator>> (std::istream &in, const Trapeze<T> &point){
    for (int i = 0; i < 4; i++)
        in >> point.Vertexs[i];
}
```

list.h

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


namespace containers {

   template<class T, class Allocator = std::allocator<T>>
   class list {
   private:
      struct element;
      size_t size = 0;
   public:
      list() = default;

      class forward_iterator {
      public:
         using value_type = T;
         using reference = value_type& ;
         using pointer = value_type* ;
         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 list;
      };

      forward_iterator begin();
      forward_iterator end();
      void push_back(const T& value);
      void push_front(const T& value);
      T& front();
      T& back();
      void pop_back();
      void pop_front();
      size_t length();
      bool empty();
      void delete_by_it(forward_iterator d_it);
      void delete_by_number(size_t N);
      void insert_by_it(forward_iterator ins_it, T& value);
      void insert_by_number(size_t N, T& value);
      list& operator=(list& other);
      T& operator[](size_t index);
   private:
      using allocator_type = typename Allocator::template
rebind<element>::other;

      struct deleter {
      private:
         allocator_type* allocator_;
      public:
         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);
```

```
            }
        }

    };

    using unique_ptr = std::unique_ptr<element, deleter>;
    struct element {
        T value;
        unique_ptr next_element = { nullptr, deleter{nullptr} };
        element* prev_element = 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 list<T, Allocator>::forward_iterator list<T, Allocator>::begin()
{
    return forward_iterator(first.get());
}

template<class T, class Allocator>
typename list<T, Allocator>::forward_iterator list<T, Allocator>::end() {
    return forward_iterator(nullptr);
}
template<class T, class Allocator>
size_t list<T, Allocator>::length() {
    return size;
}
template<class T, class Allocator>
bool list<T, Allocator>::empty() {
    return length() == 0;
}

template<class T, class Allocator>
void list<T, Allocator>::push_back(const T& value) {
    element* result = this->allocator_.allocate(1);
    std::allocator_traits<allocator_type>::construct(this->allocator_,
result, value);
    if (!size) {
        first = unique_ptr(result, deleter{ &this->allocator_ });
        tail = first.get();
        size++;
        return;
    }
    tail->next_element = unique_ptr(result, deleter{ &this->allocator_ });
    element* temp =  tail;
    tail = tail->next_element.get();
    tail->prev_element = temp;
    size++;
}

template<class T, class Allocator>
void list<T, Allocator>::push_front(const T& value) {
    size++;
    element* result = this->allocator_.allocate(1);
    std::allocator_traits<allocator_type>::construct(this->allocator_,
result, value);
    unique_ptr tmp = std::move(first);
    first = unique_ptr(result, deleter{ &this->allocator_ });
    first->next_element = std::move(tmp);
    if(first->next_element != nullptr)
```

```
        first->next_element->prev_element = first.get();
    if (size == 1) {
        tail = first.get();
    }
    if (size == 2) {
        tail = first->next_element.get();
    }
}

template<class T, class Allocator>
void list<T, Allocator>::pop_front() {
    if (size == 0) {
        throw std::logic_error("can`t pop from empty list");
    }
    if (size == 1) {
        first = nullptr;
        tail = nullptr;
        size--;
        return;
    }
    unique_ptr tmp = std::move(first->next_element);
    first = std::move(tmp);
    first->prev_element = nullptr;
    size--;
}

template<class T, class Allocator>
void list<T, Allocator>::pop_back() {
    if (size == 0) {
        throw std::logic_error("can`t pop from empty list");
    }
    if (tail->prev_element){
        element* tmp = tail->prev_element;
        tail->prev_element->next_element = nullptr;
        tail = tmp;
    }
    else{
        first = nullptr;
        tail = nullptr;
    }
    size--;
}


template<class T, class Allocator>
T& list<T, Allocator>::front() {
    if (size == 0) {
        throw std::logic_error("list is empty");
    }
    return first->value;
}

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

```
      size = other.size;
      first = std::move(other.first);
   }

   template<class T, class Allocator>
   void list<T, Allocator>::delete_by_it(containers::list<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()) {
         this->pop_front();
         return;
      }
      if (d_it.it_ptr == tail) {
         this->pop_back();
         return;
      }

      if (d_it.it_ptr == nullptr) throw std::logic_error("out of broders");
      auto temp = d_it.it_ptr->prev_element;
      unique_ptr temp1 = std::move(d_it.it_ptr->next_element);
      d_it.it_ptr->prev_element->next_element = std::move(temp1);
      d_it.it_ptr = d_it.it_ptr->prev_element;
      d_it.it_ptr->next_element->prev_element = temp;

      size--;
   }

   template<class T, class Allocator>
   void list<T, Allocator>::delete_by_number(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 list<T, Allocator>::insert_by_it(containers::list<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()) {
         this->push_front(value);
         return;
      }
      if(ins_it.it_ptr == nullptr){
         this->push_back(value);
         return;
      }

      tmp->prev_element = ins_it.it_ptr->prev_element;
      ins_it.it_ptr->prev_element = tmp;
      tmp->next_element = unique_ptr(ins_it.it_ptr, deleter{ &this-
>allocator_ });
      tmp->prev_element->next_element = unique_ptr(tmp, deleter{ &this-
>allocator_ });

      size++;
   }

   template<class T, class Allocator>
   void list<T, Allocator>::insert_by_number(size_t N, T& value) {
```

```
      forward_iterator it = this->begin();
      if (N >= this->length())
         it = this->end();
      else
         for (size_t i = 0; i < N; ++i) {
            ++it;
         }
      this->insert_by_it(it, value);
   }
   template<class T, class Allocator>
   typename list<T,Allocator>::forward_iterator list<T,
Allocator>::element::next() {
      return forward_iterator(this->next_element.get());
   }

   template<class T, class Allocator>
   list<T, Allocator>::forward_iterator::forward_iterator(containers::list<T,
Allocator>::element *ptr) {
      it_ptr = ptr;
   }

   template<class T, class Allocator>
   T& list<T, Allocator>::forward_iterator::operator*() {
      return this->it_ptr->value;
   }
   template<class T, class Allocator>
   T& list<T, Allocator>::operator[](size_t index) {
      if (index < 0 || index >= size) {
         throw std::out_of_range("out of list's borders");
      }
      forward_iterator it = this->begin();
      for (size_t i = 0; i < index; i++) {
         it++;
      }
      return *it;
   }

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

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

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

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

allocator.h

```cpp
#pragma once

#include <cstdlib>
#include <iostream>
#include <type_traits>
#include <queue>

namespace allocators {

    template<class T, size_t ALLOC_SIZE>//ALLOC_SIZE – размер, который
требуется выделить
    struct my_allocator {

    private:
        char* pool_begin; //указатель на начало хранилища
        char* pool_end;//указатель на конец хранилища
        char* pool_tail;//указатель на конец заполненного пространства
        std::queue<char*> free_blocks;
    public:
        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);

    };

    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()) {//ищем свободное место в райне отданном
пространстве
                char* ptr = free_blocks.front();
                free_blocks.pop();
                return reinterpret_cast<T*>(ptr);
            }
            std::cout<<"Bad Alloc"<<std::endl;
            throw std::bad_alloc();
```

```
        }
        T* result = reinterpret_cast<T*>(pool_tail);//приведение к типу
        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(reinterpret_cast<char*>(ptr));
    }

}
```

quenue.h

```
#pragma once

#include <memory>

#include "allocators.h"

namespace oop
{
    template<typename Q>
    class queue_forward_iterator
    {

    };

    template <typename T, typename TBaseAllocator = std::allocator<T>>
    class queue
    {

        struct node;

        using allocator = typename
std::allocator_traits<TBaseAllocator>::template rebind_alloc<node>;


        struct deleter
        {
            explicit deleter(allocator& al) noexcept
                : al_(al)
            {}

            void operator()(node* ptr)
            {
                std::allocator_traits<allocator>::destroy(al_, ptr);
                al_.deallocate(ptr, 1);
            }

        private:
            allocator& al_;
        };


        struct node
```

```
        {
            T                                   value;
            std::shared_ptr<node> next;

            explicit node(const T& v, deleter& d) noexcept
                : value(v)
                , next(nullptr, d)
            {}
        };

    public:
        queue()
            : deleter_(al_)
            , first_(nullptr, deleter_)
            , last_(nullptr)
            , size_(0)
        {}

        void pop()
        {
            if (first_)
            {
                first_ = std::move(first_->next);
            }

            --size_;
        }

        void push(const T& v)
        {
            node* obj = al_.allocate(1);
            std::allocator_traits<allocator>::construct(al_, obj, v,
deleter_);
            if (last_)
            {
                last_->next.reset(obj);
            }
            else
            {
                first_.reset(obj);
            }
            last_ = obj;

            ++size_;
        }

        [[nodiscard]] T& top()
        {
            return first_->value;
        }

        [[nodiscard]] size_t size() const noexcept
        {
            return size_;
        }

    private:
        allocator al_;
        deleter   deleter_;

        std::shared_ptr<node> first_;
        node*                           last_;
        size_t                          size_;

        friend struct node;
```

```
    };
}
```

4. Результаты выполнения тестов

1. Добавить фигуру в список
2. Удалить фигуру
3. Вывести фигуру
4. Вывести все фигуры
5. Вывести кол-во фигур чья площаль больше чем ...

1
1 0
0 0
0 0
0 0
1. Добавить фигуру в начало списка
2. Добавить фигуру в конец списка
3. Добавить фигуру по индексу
1
1. Добавить фигуру в список
2. Удалить фигуру
3. Вывести фигуру
4. Вывести все фигуры
5. Вывести кол-во фигур чья площаль больше чем ...
1
2 0
0 0
0 0
0 0
1. Добавить фигуру в начало списка
2. Добавить фигуру в конец списка
3. Добавить фигуру по индексу
2
1. Добавить фигуру в список
2. Удалить фигуру
3. Вывести фигуру
4. Вывести все фигуры
5. Вывести кол-во фигур чья площаль больше чем ...
1
1 2
2 3
3 4
5 0

1. Добавить фигуру в начало списка
2. Добавить фигуру в конец списка
3. Добавить фигуру по индексу
3
1
1. Добавить фигуру в список
2. Удалить фигуру
3. Вывести фигуру
4. Вывести все фигуры
5. Вывести кол-во фигур чья площаль больше чем ...
4
Trapeze: [1, 0] [0, 0] [0, 0] [0, 0]

Trapeze: [1, 2] [2, 3] [3, 4] [5, 0]

Trapeze: [2, 0] [0, 0] [0, 0] [0, 0]

1. Добавить фигуру в список
2. Удалить фигуру
3. Вывести фигуру
4. Вывести все фигуры
5. Вывести кол-во фигур чья площаль больше чем ...
1
3 0
0 0
0 0
0 0
1. Добавить фигуру в начало списка
2. Добавить фигуру в конец списка
3. Добавить фигуру по индексу
2
1. Добавить фигуру в список
2. Удалить фигуру
3. Вывести фигуру
4. Вывести все фигуры
5. Вывести кол-во фигур чья площаль больше чем ...
4
Trapeze: [1, 0] [0, 0] [0, 0] [0, 0]

Trapeze: [1, 2] [2, 3] [3, 4] [5, 0]

Trapeze: [2, 0] [0, 0] [0, 0] [0, 0]

Trapeze: [3, 0] [0, 0] [0, 0] [0, 0]

1. Добавить фигуру в список
2. Удалить фигуру
3. Вывести фигуру
4. Вывести все фигуры
5. Вывести кол-во фигур чья площаль больше чем ...
5
0
1
1. Добавить фигуру в список
2. Удалить фигуру
3. Вывести фигуру
4. Вывести все фигуры
5. Вывести кол-во фигур чья площаль больше чем ...
2
1. Удалить фигуру в начале списка
2. Удалить фигуру в конце списка
3. Удалить фигуру по индексу
1
1. Добавить фигуру в список
2. Удалить фигуру
3. Вывести фигуру
4. Вывести все фигуры
5. Вывести кол-во фигур чья площаль больше чем ...
2
1. Удалить фигуру в начале списка
2. Удалить фигуру в конце списка
3. Удалить фигуру по индексу
2
1. Добавить фигуру в список
2. Удалить фигуру
3. Вывести фигуру
4. Вывести все фигуры
5. Вывести кол-во фигур чья площаль больше чем ...
2
1. Удалить фигуру в начале списка
2. Удалить фигуру в конце списка
3. Удалить фигуру по индексу
31
1. Добавить фигуру в список
2. Удалить фигуру
3. Вывести фигуру
4. Вывести все фигуры
5. Вывести кол-во фигур чья площаль больше чем ...
4
Trapeze: [1, 2] [2, 3] [3, 4] [5, 0]

Trapeze: [2, 0] [0, 0] [0, 0] [0, 0]

Trapeze: [3, 0] [0, 0] [0, 0] [0, 0]

1. Добавить фигуру в список
2. Удалить фигуру
3. Вывести фигуру
4. Вывести все фигуры
5. Вывести кол-во фигур чья площаль больше чем ...
2
1. Удалить фигуру в начале списка
2. Удалить фигуру в конце списка
3. Удалить фигуру по индексу
3
1
1. Добавить фигуру в список
2. Удалить фигуру
3. Вывести фигуру
4. Вывести все фигуры
5. Вывести кол-во фигур чья площаль больше чем ...
4
Trapeze: [1, 2] [2, 3] [3, 4] [5, 0]

Trapeze: [2, 0] [0, 0] [0, 0] [0, 0]

Trapeze: [3, 0] [0, 0] [0, 0] [0, 0]

1. Добавить фигуру в список
2. Удалить фигуру
3. Вывести фигуру
4. Вывести все фигуры
5. Вывести кол-во фигур чья площаль больше чем ...
2
1. Удалить фигуру в начале списка
2. Удалить фигуру в конце списка
3. Удалить фигуру по индексу
1
1. Добавить фигуру в список
2. Удалить фигуру
3. Вывести фигуру
4. Вывести все фигуры
5. Вывести кол-во фигур чья площаль больше чем ...
4
Trapeze: [2, 0] [0, 0] [0, 0] [0, 0]

Trapeze: [3, 0] [0, 0] [0, 0] [0, 0]

1. Добавить фигуру в список
2. Удалить фигуру
3. Вывести фигуру
4. Вывести все фигуры
5. Вывести кол-во фигур чья площаль больше чем ...

2
1. Удалить фигуру в начале списка
2. Удалить фигуру в конце списка
3. Удалить фигуру по индексу
3
1
1. Добавить фигуру в список
2. Удалить фигуру
3. Вывести фигуру
4. Вывести все фигуры
5. Вывести кол-во фигур чья площаль больше чем ...

4
Trapeze: [3, 0] [0, 0] [0, 0] [0, 0]

1. Добавить фигуру в список
2. Удалить фигуру
3. Вывести фигуру
4. Вывести все фигуры
5. Вывести кол-во фигур чья площаль больше чем ...

5. Объяснение результатов работы программы

Программа выводит меню со всеми применимыми к фигурам функциями – вставкой, удалением и выводом фигур из трех различных мест, а также подсчетом фигур с площадью большей чем заданное число. Функционально программа не изменилась, однако для реализованного ранее списка был написан аллокатор, который более грамотно распоряжается памятью, отведенной для хранения списка фигур.

6. Вывод

С помощью пользовательских аллокаторов программист может более эффективно распоряжаться отданной для хранения фигур памятью, сам следить за процессом выделения и очистки памяти, конструирования и деконструирования объектов.