МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

ЛАБОРАТОРНАЯ РАБОТА №6

по курсу “Объектно-ориентированное программирование”

I семестр, 2021/22 учебный год

Студент: Недосекин Максим Александрович, группа М80-207Б-20

Преподаватель: Дорохов Евгений Павлович, каф. 806

Задание:

Используя структуру данных, разработанную для лабораторной работы №7, спроектировать и разработать аллокатор памяти для динамической структуры данных. Целью построения аллокатора является минимизация вызова операции `malloc`.

Аллокатор должен выделять большие блоки памяти для хранения фигур и при создании новых фигур-объектов выделять место под объекты в этой памяти.

Аллокатор должен хранить списки использованных/свободных блоков. Для хранения списка свободных блоков нужно применять динамическую структуру данных (контейнер 2-го уровня, согласно варианту задания).

Для вызова аллокатора должны быть переопределены операторы `new` и `delete` у классов-фигур.

Вариант №18:

- Фигура: Пятиугольник (Pentagon)
- Контейнер первого уровня: Бинарное дерево (TBinaryTree)
- Контейнер второго уровня: Стэк (TStack)

Описание программы:

Исходный код разделён на 16 файлов:

- figure.h – описание класса фигуры
- point.h – описание класса точки
- point.cpp – реализация класса точки
- trapezoid.h – описание класса трапеция
- trapezoid.cpp – реализация класса трапеция
- TBinaryTreeItem.h – описание элемента бинарного дерева
- TBinaryTreeItem.cpp – реализация элемента бинарного дерева
- TBinaryTree.h – описание бинарного дерева
- TBinaryTree.cpp – реализация бинарного дерева
- main.cpp – основная программа
- Iterator.h – реализация итератора по бинарному дереву
- HStackItem.h – описание класса элемента стека
- HStackItem.cpp – реализация класса элемента стека
- TStack.h – описание стека
- TStack.cpp – реализация класса стека
- TAllocatorBlock.h – реализация аллокатора по заданию

Дневник отладки: При выполнении работы ошибок выявлено не было.

Вывод:

Я познакомился с аллокаторами, а также поработал с классами и контейнерами, умными указателями, шаблонами и с итераторами. Мною была написана программа с собственной реализацией контейнера в виде бинарного дерева с содержанием объектов фигур, которая может вводить/выводить содержимое контейнера и удалить их при надобности. Так я смог понять, как устроена работа аллокатора и преимущества его собственной реализации с выделением памяти, что даёт возможность самому управлять памятью.

Исходный код:

`point.h:`

```
#ifndef POINT_H
#define POINT_H

#include <iostream>

class Point {
public:
  Point();
  Point(std::istream &is);
  Point(double x, double y);
  friend bool operator == (Point& p1, Point& p2);
  friend class Trapezoid;
  double dist(Point& other);
  double X();
  double Y();
  friend std::istream& operator>>(std::istream& is, Point& p);
  friend std::ostream& operator<<(std::ostream& os, Point& p);

private:
  double x;
  double y;
};

#endif
```

`point.cpp:`

```
#include "point.h"

#include <cmath>

Point::Point() : x(0.0), y(0.0) {}

Point::Point(double x, double y) : x(x), y(y) {}

Point::Point(std::istream &is) {
  is >> x >> y;
}

double Point::dist(Point& other) {
   double dx = (other.x - x);
   double dy = (other.y - y);
   return std::sqrt(dx*dx + dy*dy);
```

```
}

double Point::X() {
  return x;
};
double Point::Y() {
  return y;
};

std::istream& operator>>(std::istream& is, Point& p) {
  is >> p.x >> p.y;
  return is;
}

std::ostream& operator<<(std::ostream& os, Point& p) {
  os << "(" << p.x << ", " << p.y << ")";
  return os;
}

bool operator == (Point &p1, Point& p2) {
  return (p1.x == p2.x && p1.y == p2.y);
}
```

figure.h:

```
#ifndef FIGURE_H
#define FIGURE_H
#include <memory>
#include "point.h"

class Figure {
public:
    virtual double Area() = 0;
    virtual void Print(std::ostream &os) = 0;
    virtual size_t VertexesNumber() = 0;
    virtual ~Figure() {};
};

#endif
```

trapezoid.h:

```
#ifndef PENTAGON_H
#define PENTAGON_H

#include "figure.h"
#include <iostream>


class Trapezoid : public Figure {
    public:
    Trapezoid(std::istream &InputStream);
    Trapezoid();
    double GetArea();
    size_t VertexesNumber();
    double Area();
    void Print(std::ostream &OutputStream);
    friend bool operator == (Trapezoid& p1, Trapezoid& p2);
    friend std::ostream& operator << (std::ostream& os, Trapezoid& p);
    virtual ~Trapezoid();
    double area;

    private:
    Point a;
    Point b;
```

```
    Point c;
    Point d;
};
#endif
```

pentagon.cpp:

```
#include "trapezoid.h"
#include <cmath>

    Trapezoid::Trapezoid() {
        a.X() == 0.0; a.Y() == 0.0;
        b.X() == 0.0; b.Y() == 0.0;
        c.X() == 0.0; c.Y() == 0.0;
        d.X() == 0.0; d.Y() == 0.0;
    }

    Trapezoid::Trapezoid(std::istream &InputStream)
   {
       InputStream >> a;
       InputStream >> b;
       InputStream >> c;
       InputStream >> d;
       std:: cout << "Trapezoid that you wanted to create has been created" << std:: endl;
   }

  void Trapezoid::Print(std::ostream &OutputStream) {
      OutputStream << "Trapezoid: ";
      OutputStream << a << " " << b << " " << c << " " << d << std:: endl;

   }

   size_t Trapezoid::VertexesNumber() {
       size_t number = 4;
       return number;
    }

    double Trapezoid::Area() {
     double k = (a.Y() - d.Y()) / (a.X() - d.X());
     double m = a.Y() - k * a.X();
     double h = abs(b.Y() - k * b.X() - m) / sqrt(1 + k * k);
     return 0.5 * (a.dist(d) + b.dist(c)) * h;
    }

    double Trapezoid:: GetArea() {
       return area;
    }

    Trapezoid::~Trapezoid() {
          std:: cout << "My friend, your trapezoid has been deleted" << std:: endl;
      }

    bool operator == (Trapezoid& p1, Trapezoid& p2){
        if(p1.a == p2.a && p1.b == p2.b && p1.c == p2.c && p1.d == p2.d) {
            return true;
        }
        return false;
    }

    std::ostream& operator << (std::ostream& os, Trapezoid& p){
    os << "Trapezoid: ";
    os << p.a << p.b << p.c << p.d;
    os << std::endl;
    return os;
}
```

TBinaryTreeItem.h:

```
#include "TBinaryTreeItem.h"

template <class T>
TBinaryTreeItem<T>::TBinaryTreeItem(const T &pentagon) {
   this->pentagon = pentagon;
   this->left = this->right = NULL;
   this->counter = 1;
}

template <class T>
TBinaryTreeItem<T>::TBinaryTreeItem(const TBinaryTreeItem<T> &other) {
   this->pentagon = other.pentagon;
   this->left = other.left;
   this->right = other.right;
   this->counter = other.counter;
}

template <class T>
T& TBinaryTreeItem<T>::GetPentagon() {
   return this->pentagon;
}

template <class T>
void TBinaryTreeItem<T>::SetPentagon(const T& pentagon){
   this->pentagon = pentagon;
}

template <class T>
std::shared_ptr<TBinaryTreeItem<T>> TBinaryTreeItem<T>::GetLeft(){
   return this->left;
}

template <class T>
std::shared_ptr<TBinaryTreeItem<T>> TBinaryTreeItem<T>::GetRight(){
   return this->right;
}

template <class T>
void TBinaryTreeItem<T>::SetLeft(std::shared_ptr<TBinaryTreeItem<T>> item) {
   if (this != NULL){
      this->left = item;
   }
}

template <class T>
void TBinaryTreeItem<T>::SetRight(std::shared_ptr<TBinaryTreeItem<T>> item) {
   if (this != NULL){
      this->right = item;
   }
}

template <class T>
void TBinaryTreeItem<T>::IncreaseCounter() {
   if (this != NULL){
      counter++;
   }
}

template <class T>
void TBinaryTreeItem<T>::DecreaseCounter() {
   if (this != NULL){
      counter--;
   }
```

```
}

template <class T>
int TBinaryTreeItem<T>::ReturnCounter() {
   return this->counter;
}

template <class T>
TBinaryTreeItem<T>::~TBinaryTreeItem() {
   std::cout << "Destructor TBinaryTreeItem was called\n";
}

template <class T>
std::ostream &operator<<(std::ostream &os, TBinaryTreeItem<T> &obj)
{
   os << "Item: " << obj.GetPentagon() << std::endl;
   return os;
}

#include "trapezoid.h"
template class TBinaryTreeItem<Trapezoid>;
template std::ostream& operator<<(std::ostream& os, TBinaryTreeItem<Trapezoid> &obj);
```

TBinaryTreeItem.cpp:

```
#include "TBinaryTreeItem.h"

template <class T>
TBinaryTreeItem<T>::TBinaryTreeItem(const T &pentagon) {
   this->pentagon = pentagon;
   this->left = this->right = NULL;
   this->counter = 1;
}

template <class T>
TBinaryTreeItem<T>::TBinaryTreeItem(const TBinaryTreeItem<T> &other) {
   this->pentagon = other.pentagon;
   this->left = other.left;
   this->right = other.right;
   this->counter = other.counter;
}

template <class T>
T& TBinaryTreeItem<T>::GetPentagon() {
   return this->pentagon;
}

template <class T>
void TBinaryTreeItem<T>::SetPentagon(const T& pentagon){
   this->pentagon = pentagon;
}

template <class T>
std::shared_ptr<TBinaryTreeItem<T>> TBinaryTreeItem<T>::GetLeft(){
   return this->left;
}

template <class T>
std::shared_ptr<TBinaryTreeItem<T>> TBinaryTreeItem<T>::GetRight(){
   return this->right;
}

template <class T>
void TBinaryTreeItem<T>::SetLeft(std::shared_ptr<TBinaryTreeItem<T>> item) {
   if (this != NULL){
```

```
        this->left = item;
    }
}

template <class T>
void TBinaryTreeItem<T>::SetRight(std::shared_ptr<TBinaryTreeItem<T>> item) {
    if (this != NULL){
        this->right = item;
    }
}

template <class T>
void TBinaryTreeItem<T>::IncreaseCounter() {
    if (this != NULL){
        counter++;
    }
}

template <class T>
void TBinaryTreeItem<T>::DecreaseCounter() {
    if (this != NULL){
        counter--;
    }
}

template <class T>
int TBinaryTreeItem<T>::ReturnCounter() {
    return this->counter;
}

template <class T>
TBinaryTreeItem<T>::~TBinaryTreeItem() {
    std::cout << "Destructor TBinaryTreeItem was called\n";
}

template <class T>
std::ostream &operator<<(std::ostream &os, TBinaryTreeItem<T> &obj)
{
    os << "Item: " << obj.GetPentagon() << std::endl;
    return os;
}

#include "trapezoid.h"
template class TBinaryTreeItem<Trapezoid>;
template std::ostream& operator<<(std::ostream& os, TBinaryTreeItem<Trapezoid> &obj);
```

TBinaryTree.h:

```
#ifndef TBINARYTREE_H
#define TBINARYTREE_H
#include "TBinaryTreeItem.h"
template <class T>

class TBinaryTree {
public:
TBinaryTree();
TBinaryTree(const TBinaryTree<T> &other);
void Push(T &pentagon);
std::shared_ptr<TBinaryTreeItem<T>> Pop(std::shared_ptr<TBinaryTreeItem<T>> root, T &pentagon);
T& GetItemNotLess(double area, std::shared_ptr<TBinaryTreeItem<T>> root);
void Clear();
bool Empty();
int Count(double minArea, double maxArea);
template <class A>
```

```
friend std::ostream& operator<<(std::ostream& os, TBinaryTree<A>& tree);
virtual ~TBinaryTree();
std::shared_ptr<TBinaryTreeItem<T>> root;
};
#endif
```

TBinaryTree.cpp:

```
#include "TBinaryTree.h"

template <class T>
TBinaryTree<T>::TBinaryTree () {
   root = NULL;
}

template <class T>
std::shared_ptr<TBinaryTreeItem<T>> copy (std::shared_ptr<TBinaryTreeItem<T>> root) {
   if (!root) {
      return NULL;
   }
   std::shared_ptr<TBinaryTreeItem<T>> root_copy(new TBinaryTreeItem<T>(root->GetPentagon()));
   root_copy->SetLeft(copy(root->GetLeft()));
   root_copy->SetRight(copy(root->GetRight()));
   return root_copy;
}

template <class T>
TBinaryTree<T>::TBinaryTree (const TBinaryTree<T> &other) {
   root = copy(other.root);
}

template <class T>
void Print (std::ostream& os, std::shared_ptr<TBinaryTreeItem<T>> node){
   if (!node){
      return;
   }
   if(node->GetLeft()){
      os << node->GetPentagon().GetArea() << ": [";
      Print (os, node->GetLeft());
      if (node->GetRight()){
         if (node->GetRight()){
            os << ", ";
            Print (os, node->GetRight());
         }
      }
      os << "]";
   } else if (node->GetRight()) {
      os << node->GetPentagon().GetArea() << ": [";
      Print (os, node->GetRight());
      if (node->GetLeft()){
         if (node->GetLeft()){
            os << ", ";
            Print (os, node->GetLeft());
         }
      }
      os << "]";
   }
   else {
      os << node->GetPentagon().GetArea();
   }
}

template <class T>
std::ostream& operator<< (std::ostream& os, TBinaryTree<T>& tree){
   Print(os, tree.root);
```

```
    os << "\n";
    return os;
}

template <class T>
void TBinaryTree<T>::Push (T &pentagon) {
    if (root == NULL) {
    std::shared_ptr<TBinaryTreeItem<T>> help(new TBinaryTreeItem<T>(pentagon));
    root = help;
    }
    else if (root->GetPentagon() == pentagon) {
        root->IncreaseCounter();
    }
    else {
        std::shared_ptr <TBinaryTreeItem<T>> parent = root;
        std::shared_ptr <TBinaryTreeItem<T>> current;
        bool childInLeft = true;
        if (pentagon.GetArea() < parent->GetPentagon().GetArea()) {
            current = root->GetLeft();
        }
        else if (pentagon.GetArea() > parent->GetPentagon().GetArea()) {
            current = root->GetRight();
            childInLeft = false;
        }
        while (current != NULL) {
            if (current->GetPentagon() == pentagon) {
                current->IncreaseCounter();
            }
            else {
            if (pentagon.GetArea() < current->GetPentagon().GetArea()) {
                parent = current;
                current = parent->GetLeft();
                childInLeft = true;
                }
            else if (pentagon.GetArea() > current->GetPentagon().GetArea()) {
                parent = current;
                current = parent->GetRight();
                childInLeft = false;
            }
        }
    }
        std::shared_ptr <TBinaryTreeItem<T>> item (new TBinaryTreeItem<T>(pentagon));
        current = item;
        if (childInLeft == true) {
            parent->SetLeft(current);
        }
        else {
            parent->SetRight(current);
        }
    }
}

template <class T>
std::shared_ptr <TBinaryTreeItem<T>> FMRST(std::shared_ptr <TBinaryTreeItem<T>> root) {
    if (root->GetLeft() == NULL) {
        return root;
    }
    return FMRST(root->GetLeft());
}

template <class T>
std::shared_ptr <TBinaryTreeItem<T>> TBinaryTree<T>:: Pop(std::shared_ptr <TBinaryTreeItem<T>> root, T &pentagon) {
    if (root == NULL) {
        return root;
    }
    else if (pentagon.GetArea() < root->GetPentagon().GetArea()) {
```

```
        root->SetLeft(Pop(root->GetLeft(), pentagon));
    }
    else if (pentagon.GetArea() > root->GetPentagon().GetArea()) {
        root->SetRight(Pop(root->GetRight(), pentagon));
    }
    else {
        //first case of deleting - we are deleting a list
        if (root->GetLeft() == NULL && root->GetRight() == NULL) {
            root = NULL;
            return root;
        }
        //second case of deleting - we are deleting a verex with only one child
        else if (root->GetLeft() == NULL && root->GetRight() != NULL) {
            std::shared_ptr <TBinaryTreeItem<T>> pointer = root;
            root = root->GetRight();
            return root;
        }
        else if (root->GetRight() == NULL && root->GetLeft() != NULL) {
            std::shared_ptr <TBinaryTreeItem<T>> pointer = root;
            root = root->GetLeft();
            return root;
        }
        //third case of deleting
        else {
            std::shared_ptr <TBinaryTreeItem<T>> pointer = FMRST(root->GetRight());
            root->GetPentagon().area = pointer->GetPentagon().GetArea();
            root->SetRight(Pop(root->GetRight(), pointer->GetPentagon()));
        }
    }
    return root;
}

template <class T>
void RecursiveCount(double minArea, double maxArea, std::shared_ptr<TBinaryTreeItem<T>> current, int& ans) {
    if (current != NULL) {
        RecursiveCount(minArea, maxArea, current->GetLeft(), ans);
        RecursiveCount(minArea, maxArea, current->GetRight(), ans);
        if (minArea <= current->GetPentagon().GetArea() && current->GetPentagon().GetArea() < maxArea) {
            ans += current->ReturnCounter();
        }
    }
}

template <class T>
int TBinaryTree<T>::Count(double minArea, double maxArea) {
    int ans = 0;
    RecursiveCount(minArea, maxArea, root, ans);
    return ans;
}

template <class T>
T& TBinaryTree<T>::GetItemNotLess(double area, std::shared_ptr <TBinaryTreeItem<T>> root) {
    if (root->GetPentagon().GetArea() >= area) {
        return root->GetPentagon();
    }
    else {
        return GetItemNotLess(area, root->GetRight());
    }
}

template <class T>
void RecursiveClear(std::shared_ptr <TBinaryTreeItem<T>> current){
    if (current!= NULL){
        RecursiveClear(current->GetLeft());
        RecursiveClear(current->GetRight());
            current = NULL;
```

```
    }
}

template <class T>
void TBinaryTree<T>::Clear(){
    RecursiveClear(root);
    root = NULL;
}

template <class T>
bool TBinaryTree<T>::Empty() {
     if (root == NULL) {
         return true;
     }
     return false;
}

template <class T>
TBinaryTree<T>::~TBinaryTree() {
    Clear();
    std:: cout << "Your tree has been deleted" << std:: endl;
}

#include "trapezoid.h"
template class TBinaryTree<Trapezoid>;
template std::ostream& operator<<(std::ostream& os, TBinaryTree<Trapezoid>& stack);
```

TIterator.h:

```
#ifndef TITERATOR_H
#define TITERATOR_H
#include <iostream>
#include <memory>

template <class T, class A>
class TIterator {
public:
TIterator(std::shared_ptr<T> iter) {
    node_ptr = iter;
}
A& operator*() {
    return node_ptr->GetTrapezoid();
}

void GoToLeft() { //переход к левому поддереву, если существует
    if (node_ptr == NULL) {
        std:: cout << "Root does not exist" << std:: endl;
    }
    else {
        node_ptr = node_ptr->GetLeft();
    }
}
void GoToRight() { //переход к правому поддереву, если существует
    if (node_ptr == NULL) {
        std:: cout << "Root does not exist" << std:: endl;
    }
    else {
        node_ptr = node_ptr->GetRight();
    }
}
bool operator == (TIterator &iterator) {
    return node_ptr == iterator.node_ptr;
}
bool operator != (TIterator &iterator) {
    return !(*this == iterator);
}
```

```
private:
    std::shared_ptr<T> node_ptr;
};
#endif
```

main.cpp:

```
#include <iostream>
#include "trapezoid.h"
#include "TBinaryTree.h"
#include "TBinaryTreeItem.h"
int main () {
    //lab1
    Trapezoid a (std::cin);
    std:: cout << "The area of your figure is : " << a.Area() << std:: endl;

    Trapezoid b (std::cin);
    std:: cout << "The area of your figure is : " << b.Area() << std:: endl;

    Trapezoid c (std::cin);
    std:: cout << "The area of your figure is : " << c.Area() << std:: endl;

    //lab4
    TBinaryTree<Trapezoid> tree;
    std:: cout << "Is tree empty? " << tree.Empty() << std:: endl;
    std:: cout << "And now, is tree empty? " << tree.Empty() << std:: endl;
    tree.Push(a);
    tree.Push(b);
    tree.Push(c);
    std:: cout << "The number of figures with area in [minArea, maxArea] is: " << tree.Count(0, 100000) << std:: endl;
    std:: cout << "The result of searching the same-figure-counter is: " << tree.root->ReturnCounter() << std:: endl;
    std:: cout << "The result of function named GetItemNotLess is: " << tree.GetItemNotLess(0, tree.root) << std:: endl;
    std:: cout << tree << std:: endl;
    tree.root = tree.Pop(tree.root, a);
    std:: cout << tree << std:: endl;
    return 0;
}
```

HStackItem.h:

```
#ifndef HLISTITEM_H
#define HLISTITEM_H
#include <iostream>
#include "trapezoid.h"
```

```
#include <memory>

template <class T> class HStackItem {
public:
  HStackItem(const std::shared_ptr<Trapezoid> &trapezoid);
  template <class A> friend std::ostream& operator<<(std::ostream& os, HStackItem<A> &obj);
  ~HStackItem();
  std::shared_ptr<T> trapezoid;
  std::shared_ptr<HStackItem<T>> next;
};
#endif
```

HStackItem.cpp:

```
#include <iostream>
#include "HStackItem.h"



template <class T> HStackItem<T>::HStackItem(const std::shared_ptr<Trapezoid> &trapezoid) {
  this->trapezoid = trapezoid;
  this->next = nullptr;
}
template <class A> std::ostream& operator<<(std::ostream& os,HStackItem<A> &obj) {
  os << "[" << obj.trapezoid << "]" << std::endl;
  return os;
}
template <class T> HStackItem<T>::~HStackItem() {
}
```

TStack.h:

```
#ifndef HLIST_H
#define HLIST_H
#include <iostream>
#include "HStackItem.h"
#include "trapezoid.h"
#include <memory>

template <class T> class TStack {
public:
  TStack();
  int size_of_list;
  size_t Length();
```

```
  std::shared_ptr<Trapezoid>& Last();
  std::shared_ptr<Trapezoid>& GetItem(size_t idx);
  bool Empty();
  TStack(const std::shared_ptr<TStack> &other);
  void InsertLast(const std::shared_ptr<Trapezoid> &&trapezoid);
  void RemoveLast();
  void Insert(const std::shared_ptr<Trapezoid> &&trapezoid, size_t position);
  void Remove(size_t position);
  void Clear();
  template <class A> friend std::ostream& operator<<(std::ostream& os, TStack<A>& list);
  ~TStack();
private:
  std::shared_ptr<HStackItem<T>> front;
  std::shared_ptr<HStackItem<T>> back;
};
#endif
```

TStack.cpp:

```
#include <iostream>
#include "TStack.h"
#include "HStackItem.h"

template <class T> TStack<T>::TStack() {
  size_of_list = 0;
  std::shared_ptr<HStackItem<T>> front;
  std::shared_ptr<HStackItem<T>> back;
  std::cout << "Trapezoid Stack created" << std::endl;
}
template <class T> TStack<T>::TStack(const std::shared_ptr<TStack> &other){
  front = other->front;
  back = other->back;
}
template <class T> size_t TStack<T>::Length() {
  return size_of_list;
}
template <class T> bool TStack<T>::Empty() {
  return size_of_list;
}
template <class T> std::shared_ptr<Trapezoid>& TStack<T>::GetItem(size_t idx){
  int k = 0;
  std::shared_ptr<HStackItem<T>> obj = front;
```

```
  while (k != idx){
    k++;
    obj = obj->next;
  }
  return obj->trapezoid;
}
template <class T> std::shared_ptr<Trapezoid>& TStack<T>::Last() {
  return back->trapezoid;
}
template <class T> void TStack<T>::InsertLast(const std::shared_ptr<Trapezoid> &&trapezoid) {
  std::shared_ptr<HStackItem<T>> obj (new HStackItem<T>(trapezoid));
  if(size_of_list == 0) {
    front = obj;
    back = obj;
    size_of_list++;
    return;
  }
  back->next = obj;
  back = obj;
  obj->next = nullptr;
  size_of_list++;
}
template <class T> void TStack<T>::RemoveLast() {
  if (size_of_list == 0) {
    std::cout << "Trapezoid does not pop_back, because the Trapezoid List is empty" << std:: endl;
  } else {
    std::shared_ptr<HStackItem<T>> prev_del = front;
    while (prev_del->next != back) {
      prev_del = prev_del->next;
    }
    prev_del->next = nullptr;
    back = prev_del;
    size_of_list--;
    }
}
template <class T> void TStack<T>::Insert(const std::shared_ptr<Trapezoid> &&trapezoid, size_t position) {
  if (position <0) {
    std::cout << "Position < zero" << std::endl;
  } else if (position > size_of_list) {
    std::cout << " Position > size_of_list" << std::endl;
  } else {
```

```
    std::shared_ptr<HStackItem<T>> obj (new HStackItem<T>(trapezoid));
    if (position == 0) {
      front = obj;
      back = obj;
    } else {
      int k = 0;
      std::shared_ptr<HStackItem<T>> prev_insert = front;
      std::shared_ptr<HStackItem<T>> next_insert;
      while(k+1 != position) {
        k++;
        prev_insert = prev_insert->next;
      }
      next_insert = prev_insert->next;
      prev_insert->next = obj;
      obj->next = next_insert;
    }
    size_of_list++;
  }
}
template <class T> void TStack<T>::Remove(size_t position) {
  if (position > size_of_list ) {
    std:: cout << "Position " << position << " > " << "size " << size_of_list << " Not correct erase" << std::endl;
  } else if (position < 0) {
    std::cout << "Position < 0" << std::endl;
  } else {
      int k = 0;
      std::shared_ptr<HStackItem<T>> prev_erase = front;
      std::shared_ptr<HStackItem<T>> next_erase;
      std::shared_ptr<HStackItem<T>> del;
      while( k+1 != position) {
        k++;
        prev_erase = prev_erase->next;
      }
      next_erase = prev_erase->next;
      del = prev_erase->next;
      next_erase = del->next;
      prev_erase->next = next_erase;
    size_of_list--;
  }
}
template <class T> void TStack<T>::Clear() {
```

```
  std::shared_ptr<HStackItem<T>> del = front;
  std::shared_ptr<HStackItem<T>> prev_del;
  if(size_of_list !=0 ) {
    while(del->next != nullptr) {
      prev_del = del;
      del = del->next;
    }
    size_of_list = 0;
    //   std::cout << "HStackItem deleted" << std::endl;
  }
  size_of_list = 0;
  std::shared_ptr<HStackItem<T>> front;
  std::shared_ptr<HStackItem<T>> back;
}
template <class T> std::ostream& operator<<(std::ostream& os, TStack<T>& hl) {
  if (hl.size_of_list == 0) {
    os << "The trapezoid stack is empty, so there is nothing to output" << std::endl;
  } else {
    os << "Print Trapezoid Stack" << std::endl;
    std::shared_ptr<HStackItem<T>> obj = hl.front;
    while(obj != nullptr) {
      if (obj->next != nullptr) {
        os << obj->trapezoid << " " << "," << " ";
        obj = obj->next;
      } else {
        os << obj->trapezoid;
        obj = obj->next;
      }
    }
    os << std::endl;
  }
  return os;
}
template <class T> TStack<T>::~TStack() {
  std::shared_ptr<HStackItem<T>> del = front;
  std::shared_ptr<HStackItem<T>> prev_del;
  if(size_of_list !=0 ) {
    while(del->next != nullptr) {
      prev_del = del;
      del = del->next;
    }
```

```
    size_of_list = 0;
    std::cout << "Trapezoid Stack deleted" << std::endl;
  }
}
```

TAllocatorBlock.h:

```
#ifndef TALLOCATORBLOCK_H
#define TALLOCATORBLOCK_H

#include "TStack.h"
#include <memory>

class TAllocatorBlock {
    public:
        TAllocatorBlock(const size_t& size, const size_t count){
            this->size = size;
            for(int i = 0; i < count; ++i){
                unused_blocks.Insert(malloc(size));
            }
        }
        void* Allocate(const size_t& size){
            if(size != this->size){
                std::cout << "Error during allocation\n";
            }
            if(unused_blocks.Length()){
                for(int i = 0; i < 5; ++i){
                    unused_blocks.Insert(malloc(size));
                }
            }
            void* tmp = unused_blocks.GetItem(1);
            used_blocks.Insert(unused_blocks.GetItem(1));
            unused_blocks.Remove(0);
            return tmp;
        }
        void Deallocate(void* ptr){
            unused_blocks.Insert(ptr);
        }
    ~TAllocatorBlock(){
        while(used_blocks.size()){
            try{
                free(used_blocks.GetItem(1);)
```

```
            used_blocks.Remove(0);
        } catch(...){
            used_blocks.Remove(0);
        }
    }
    while(unused_blocks.size()){
        try{
            free(unused_blocks.GetItem(1);
            unused_blocks.Remove(0);
        } catch(...){
            unused_blocks.Remove(0);
        }
    }
  }

  private:
    size_t size;
    TStack <void*> used_blocks;
    TStack <void*> unused_blocks;
};

#endif
```

Результат работы:

```
The area of your figure is : 2
0 0 1 1 2 1 3 0
Trapezoid that you wanted to create has been created
The area of your figure is : 2
0 0 1 1 2 1 3 0
The area of your figure is : 2
0 0 1 1 2 1 3 0
Trapezoid that you wanted to create has been created
Trapezoid that you wanted to create has been created
The area of your figure is : 2
0 0 1 1 2 1 3 0
Trapezoid that you wanted to create has been created
The area of your figure is : 2
Is tree empty? 1
And now, is tree empty? 0
The number of figures with area in [minArea, maxArea] is: 5
The result of searching the same-figure-counter is: 5
The result of function named GetItemNotLess is: Trapezoid: (0, 0)(1, 1)(2, 1)(3, 0)
```