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

ЛАБОРАТОРНАЯ РАБОТА №8

по курсу “Объектно-ориентированное программирование”

I семестр, 2021/22 учебный год

Студент: Куценко Борис Дмитриевич

Группа: М8О-207Б-20

Преподаватель: Дорохов Евгений Павлович

Москва, 2021

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 №13:

- Фигуры: Ромб
- Контейнер первого уровня: Бинарное дерево
- Контейнер второго уровня: Динамический массив

Описание программы:

Исходный код разделён на 17 файлов:

- figure.h – описание класса фигуры
- point.h – описание класса точки
- point.cpp – реализация класса точки
- rhombus.h – описание класса ромб (наследуется от фигуры)
- rhombus.cpp – реализация класса ромб
- tbinarytreeitem.h – описание элемента бинарного дерева
- tbinarytreeitem.cpp – реализация элемента бинарного дерева
- tbinarytree.h – описание бинарного дерева
- tbinarytree.cpp – реализация бинарного дерева
- TIterator.h – реализация итератора
- tallocator.h/cpp – реализация класса аллокатора для фигуры
- tvector.h/cpp – реализация класса динамического массива для использования в аллокаторе
- tvector_item.h/cpp – реализация класса элемента динамического массива для использования в аллокаторе
- main.cpp – основная программа

Дневник отладки:

При выполнении работы ошибок выявлено не было.

Вывод:

Во воемя написания данной лабораторной работы я познакомился с понятием аллокатора и реализовал свою версию. Реализация собственного аллокатора для каждого проекта язвляется важным аспектом эффективности работы программы по памяти.

Исходный код:

point.h:

```
#ifndef POINT_H
#define POINT_H

#include <iostream>

class Point {
public:
    Point();
    Point(std::istream&);
    Point(double x, double y);
    Point(const Point& other);

    double dist(Point& other);
    friend std::istream& operator>>(std::istream& is, Point& p);
    friend std::ostream& operator<<(std::ostream&, const Point& p);
    bool operator==(const Point& other);

public:
    double x_;
    double y_;
};

#endif
```

point.cpp:

```
#include "point.h"
#include <iostream>
#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_;
}
Point::Point(const Point& other) : x_(other.x_), y_(other.y_) {}

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

```
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 Point::operator==(const Point& other) {
    return (x_ == other.x_ && y_ == other.y_);
}
```

figure.h:

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

class Figure {

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


#endif
```

rhombus.h:

```
#ifndef RHOMBUS_H
#define RHOMBUS_H

#include "figure.h"
#include "tallocator.h"


class Rhombus : public Figure {
public:
    Rhombus();
    virtual ~Rhombus();
    Rhombus(std::istream &in);
    Rhombus(const Rhombus& r);
    Rhombus(Point& x1, Point& x2, Point& x3, Point& x4);
    double Area();
    size_t VertexesNumber();
    bool IsRhombus() ;
    bool operator==( Rhombus& r);
    bool operator!=( Rhombus& r);
```

```
    Rhombus& operator=(const Rhombus& r);
    void Print(std::ostream& os);
    friend std::ostream& operator<<(std::ostream &os, const Rhombus& r);
    friend std::istream& operator>> (std::istream &in, Rhombus &r);

    void *operator new(size_t sz);
    void operator delete(void *ptr, size_t sz) noexcept;

protected:
    static TAllocator alloc;
    Point _x1, _x2, _x3, _x4;
};

#endif
```

rhombus.cpp:

```
#include "rhombus.h"
#include <string.h>


Rhombus::Rhombus(): _x1(0, 0), _x2(0, 0), _x3(0, 0), _x4(0, 0) {}
TAllocator Rhombus::alloc(sizeof(Rhombus), 10);
Rhombus::~Rhombus() {}

double Rhombus::Area() {
    return 0.5 *_x1.dist(_x3) * _x2.dist(_x4);
}

bool Rhombus::IsRhombus() {
    if (_x1.dist(_x2) == _x2.dist(_x3) && _x2.dist(_x3) == _x3.dist(_x4) &&
    _x3.dist(_x4) == _x4.dist(_x1) && _x4.dist(_x1) == _x1.dist(_x2))
        return true;
    return false;
}
Rhombus::Rhombus(Point &x1, Point &x2, Point &x3, Point &x4) : _x1(x1), _x2(x2), _x3(x3),
_x4(x4){
    if(!IsRhombus()) {
        std::cout << "ERORR:it isn't rhombus, incorrect input\n";
        exit(-1);
    }
}

size_t Rhombus::VertexesNumber() {
    return 4;
}


void Rhombus::Print(std::ostream& os) {
    os << "Rhombus: (" << _x1.x_ << ", " << _x1.y_ << ") " << '(' << _x2.x_ << ", " <<
_x2.y_ << ") "
    << '(' << _x3.x_ << ", " << _x3.y_ << ") " << '(' << _x4.x_ << ", " << _x4.y_   << ")"
<< std::endl;
}
```

```cpp
std::ostream& operator<<(std::ostream &os, const Rhombus& r)
{
    os << "Rhombus: (" << r._x1.x_ << ", " << r._x1.y_ << ") " << '(' << r._x2.x_ << ", "
<< r._x2.y_ << ") "
       << '(' << r._x3.x_ << ", " << r._x3.y_ << ") " << '(' << r._x4.x_ << ", " <<
r._x4.y_ << ")" << std::endl;
    return os;
}

std::istream &operator>>(std::istream &in, Rhombus &r) {
    in >> r._x1.x_ >> r._x1.y_>> r._x2.x_ >> r._x2.y_ >> r._x3.x_ >> r._x3.y_ >> r._x4.x_
>> r._x4.y_;
    if(!r.IsRhombus()) {
        std::cout << "ERORR:it isn't rhombus, incorrect input\n";
        exit(-1);
    }
    return in;
}

Rhombus::Rhombus(const Rhombus &r) : _x1(r._x1), _x2(r._x2), _x3(r._x3), _x4(r._x4) {}

Rhombus::Rhombus(std::istream &in) {
    in >> _x1.x_ >> _x1.y_ >> _x2.x_ >> _x2.y_ >> _x3.x_ >> _x3.y_ >> _x4.x_ >> _x4.y_;
    if (!IsRhombus())  {
        std::cout << "ERORR:it isn't rhombus, incorrect input\n";
        exit(-1);
    }
}

Rhombus &Rhombus::operator=(const Rhombus &r) {
    if (&r == this)
        return *this;
    _x1.x_ = r._x1.x_;
    _x1.y_ = r._x1.y_;
    _x2.x_ = r._x2.x_;
    _x2.y_ = r._x2.y_;
    _x3.x_ = r._x3.x_;
    _x3.y_ = r._x3.y_;
    _x4.x_ = r._x4.x_;
    _x4.y_ = r._x4.y_;
    return *this;

}

bool Rhombus::operator==( Rhombus &r)  {
    return _x1 == r._x1 && _x2 == r._x2 && _x3 == r._x3 && _x4 == r._x4;
}
bool Rhombus::operator!=(Rhombus &r) {
    return !(*this == r);
}
void *Rhombus::operator new(size_t sz) {
    return alloc.allocate(sz);
}
```

```
void Rhombus::operator delete(void *ptr, size_t sz) noexcept {
    alloc.deallocate(ptr);
}
```

tbinarytreeitem.h:

```
#ifndef ITEM_H
#define ITEM_H

#include <memory>
#include <iostream>

using std::shared_ptr;
using std::make_shared;

template <class T>
class TreeElem{
public:
    TreeElem();
    TreeElem(T rhombus);

    T& get_rhombus();
    int get_count();
    shared_ptr<TreeElem<T>> get_left() ;
    shared_ptr<TreeElem<T>> get_right();

    void set_rhombus(T& rhombus);
    void set_count(int count);
    void set_left(shared_ptr<TreeElem<T>> to_left);
    void set_right(shared_ptr<TreeElem<T>> to_right);

    virtual ~TreeElem();
private:
    shared_ptr<T> r;
    int count;
    shared_ptr<TreeElem<T>> left;
    shared_ptr<TreeElem<T>> right;
};

#endif
```

tbinarytreeitem.cpp:

```
#include "tbinarytreeitem.h"
#include <memory>


template <class T>
TreeElem<T>::TreeElem() {
    r = nullptr;
    count = 0;
    left = nullptr;
    right = nullptr;
}
```

```
template <class T>
TreeElem<T>::TreeElem(T rhombus) {
    r = make_shared<T>(rhombus);
    count = 1;
    left = nullptr;
    right = nullptr;
}

template <class T>
 T& TreeElem<T>::get_rhombus() {
    return *r;
}
template <class T>
int TreeElem<T>::get_count() {
    return count;
}
template <class T>
shared_ptr<TreeElem<T>> TreeElem<T>::get_left() {
    return left;
}
template <class T>
shared_ptr<TreeElem<T>> TreeElem<T>::get_right() {
    return right;
}

template <class T>
void TreeElem<T>::set_rhombus( T& rhombus){
    r = make_shared<T>(rhombus);
}
template <class T>
void TreeElem<T>::set_count( int count_) {
    count = count_;
}
template <class T>
void TreeElem<T>::set_left(shared_ptr<TreeElem<T>> to_left) {
    left = to_left;
}
template <class T>
void TreeElem<T>::set_right(shared_ptr<TreeElem<T>> to_right) {
    right = to_right;
}

template <class T>
TreeElem<T>::~TreeElem() {
}

#include "rhombus.h"
template class TreeElem<Rhombus>;
```

tbinarytree.h:

```
#ifndef TBINARYTREE_H
```

```h
#define TBINARYTREE_H
#include "tbinarytreeitem.h"

template <class T>
class TBinaryTree {
public:
    // Конструктор по умолчанию.
    TBinaryTree();

    void Push(T& rhombus);
    // Метод получения фигуры из контейнера.
    // Если площадь превышает максимально возможную,
    // метод должен бросить исключение std::out_of_range
    T& GetItemNotLess(double area);
    // Метод, возвращающий количество совпадающих фигур с данными параметрами
    size_t Count(T& rhombus);
    // Метод по удалению фигуры из дерева:
    // Счетчик вершины уменьшается на единицу.
    // Если счетчик становится равен 0,
    // вершина удаляется с заменой на корректный узел поддерева.
    // Если такой вершины нет, бросается исключение std::invalid_argument
    void Pop(T& rhombus);
    // Метод проверки наличия в дереве вершин
    bool Empty();
    // Оператор вывода дерева в формате вложенных списков,
    // где каждый вложенный список является поддеревом текущей вершины:
    // "S0: [S1: [S3, S4: [S5, S6]], S2]",
    // где Si - строка вида количество*площадь_фигуры
    // Пример: 1*1.5: [3*1.0, 2*2.0: [2*1.5, 1*6.4]]
    template <class A>
    friend std::ostream& operator<<(std::ostream& os, const TBinaryTree<A>& tree);
    // Метод, удаляющий все элементы контейнера,
    // но позволяющий пользоваться им.
    void Clear();
    // Деструктор
    virtual ~TBinaryTree();
private:
    shared_ptr<TreeElem<T>> root;
};
#endif
```

tbinarytree.cpp:

```cpp
#include "tbinarytree.h"


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

template <class T>
void TBinaryTree<T>::Push( T& rhombus) {
    shared_ptr<TreeElem<T>> curr = root;
```

```
    shared_ptr<TreeElem<T>> rootptr = make_shared<TreeElem<T>> (rhombus);

    if (!curr)
    {
        root = rootptr;
    }
    while (curr)
    {
        if (curr->get_rhombus() == rhombus)
        {
            curr->set_count(curr->get_count() + 1);
            return;
        }
        if (rhombus.Area() < curr->get_rhombus().Area())
            if (curr->get_left() == nullptr)
            {
                curr->set_left(rootptr);
                return;
            }
        if (rhombus.Area() >= curr->get_rhombus().Area())
            if (curr->get_right() == nullptr && !(curr->get_rhombus() == rhombus))
            {
                curr->set_right(rootptr);
                return;
            }
        if (curr->get_rhombus().Area() > rhombus.Area())
            curr = curr->get_left();
        else
            curr = curr->get_right();
    }
}

template <class T>
T& TBinaryTree<T>::GetItemNotLess(double area) {
    shared_ptr<TreeElem<T>> curr = root;
    while (curr)
    {
        if (area == curr->get_rhombus().Area())
            return curr->get_rhombus();
        if (area < curr->get_rhombus().Area())
        {
            curr = curr->get_left();
            continue;
        }
        if (area >= curr->get_rhombus().Area())
        {
            curr = curr->get_right();
            continue;
        }
    }
    throw std::out_of_range("out of range");
}

template <class T>
```

```
size_t TBinaryTree<T>::Count( T& rhombus) {
    size_t count = 0;
    shared_ptr<TreeElem<T>> curr = root;

    while (curr)
    {
        if (curr->get_rhombus() == rhombus)
            count = curr->get_count();
        if (rhombus.Area() < curr->get_rhombus().Area())
        {
            curr = curr->get_left();
            continue;
        }
        if (rhombus.Area() >= curr->get_rhombus().Area())
        {
            curr = curr->get_right();
            continue;
        }
    }
    return count;
}

template <class T>
void Pop_List(shared_ptr<TreeElem<T>> curr, shared_ptr<TreeElem<T>> parent) {
    if (parent->get_left() == curr)
        parent->set_left(nullptr);
    else
        parent->set_right(nullptr);
}
template <class T>
void Pop_Part_of_Branch(shared_ptr<TreeElem<T>> curr, shared_ptr<TreeElem<T>> parent) {
    if (parent) {
        if (curr->get_left()) {
            if (parent->get_left() == curr)
                parent->set_left(curr->get_left());

            if (parent->get_right() == curr)
                parent->set_right(curr->get_left());

            curr->set_right(nullptr);
            curr->set_left(nullptr);
            return;
        }

        if (curr->get_left() == nullptr) {
            if (parent && parent->get_left() == curr)
                parent->set_left(curr->get_right());

            if (parent && parent->get_right() == curr)
                parent->set_right(curr->get_right());

            curr->set_right(nullptr);
            curr->set_left(nullptr);
            return;
```

```
        }
    }
}
template <class T>
void Pop_Root_of_Subtree(shared_ptr<TreeElem<T>> curr, shared_ptr<TreeElem<T>> parent) {
    shared_ptr<TreeElem<T>> replace = curr->get_left();
    shared_ptr<TreeElem<T>> rep_parent = curr;
    while (replace->get_right())
    {
        rep_parent = replace;
        replace = replace->get_right();
    }

    curr->set_rhombus(replace->get_rhombus());
    curr->set_count(replace->get_count());

    if (rep_parent->get_left() == replace)
        rep_parent->set_left(nullptr);
    else
        rep_parent->set_right(nullptr);
    return;
}
template <class T>
void TBinaryTree<T>::Pop( T& rhombus) {

    shared_ptr<TreeElem<T>> curr = root;
    shared_ptr<TreeElem<T>> parent = nullptr;

    while (curr && curr->get_rhombus() != rhombus)
    {
        parent = curr;
        if (curr->get_rhombus().Area() > rhombus.Area())
            curr = curr->get_left();
        else
            curr = curr->get_right();
    }

    if (curr == nullptr)
        return;

    curr->set_count(curr->get_count() - 1);

    if(curr->get_count() <= 0)
    {
        if (curr->get_left() == nullptr && curr->get_right() == nullptr)
        {
            Pop_List(curr, parent);
            return;
        }
        if (curr->get_left() == nullptr || curr->get_right() == nullptr)
        {
            Pop_Part_of_Branch(curr, parent);
            return;
        }
```

```
        if (curr->get_left() != nullptr && curr->get_right() != nullptr)
        {
            Pop_Root_of_Subtree(curr, parent);
            return;
        }
    }
}

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

template <class T>
void Tree_out (std::ostream& os, shared_ptr<TreeElem<T>> curr) {
    if (curr)
    {
        if(curr->get_rhombus().Area() >= 0)
            os << curr->get_count() << "*" << curr->get_rhombus().Area();
        if(curr->get_left() || curr->get_right())
        {
            os << ": [";
            if (curr->get_left())
                Tree_out(os, curr->get_left());
            if(curr->get_left() && curr->get_right())
                os << ", ";
            if (curr->get_right())
                Tree_out(os, curr->get_right());
            os << "]";
        }
    }
}
template <class A>
std::ostream& operator<<(std::ostream& os, const TBinaryTree<A>& tree) {
    shared_ptr<TreeElem<A>> curr = tree.root;
    Tree_out(os, curr);
    return os;
}

template <class T>
void recursive_clear(shared_ptr<TreeElem<T>> curr){
    if(curr)
    {
        if (curr->get_left())
            recursive_clear(curr->get_left());
        curr->set_left(nullptr);
        if (curr->get_right())
            recursive_clear(curr->get_right());
        curr->set_right(nullptr);
    }
}
template <class T>
void TBinaryTree<T>::Clear() {
    if (root->get_left())
```

```
        recursive_clear(root->get_left());
    root->set_left(nullptr);
    if (root->get_right())
        recursive_clear(root->get_right());
    root->set_right(nullptr);
    root = nullptr;
}

template <class T>
TBinaryTree<T>::~TBinaryTree() {
}

#include "rhombus.h"
template class TBinaryTree<Rhombus>;
template std::ostream& operator<<(std::ostream& os, const TBinaryTree<Rhombus>& rhombus);
```

main.cpp:

```
#include "rhombus.h"
#include "tbinarytree.h"

int main()
{
    Rhombus *rhomb[4];
    for (int i = 0; i < 4; ++i) {
        std::cout << "Enter a Rhombus: ";
        Rhombus a;
        std::cin >> a;
        rhomb[i] = new Rhombus(a);

    }
    std::cout << "created\n";
    for (int i = 0; i < 4; i++) {
        std::cout << *rhomb[i] << "\n";
    }
    for (int i = 0; i < 4; ++i) {
        delete rhomb[i];
    }
    return 0;
}
```

tvector.h:

```
#ifndef TVECTOR_H
#define TVECTOR_H

#include <memory>
#include "iostream"
#include "tvector_item.h"
#include <new>

template<typename T>
class TVector {
public:
    TVector();
```

```
    TVector(size_t size);

    TVector(size_t size, T filler);

    TVector(const TVector &other);

    void InsertLast(const T &elem);

    void RemoveLast();

    const T &Last();

    T &operator[](const size_t& idx);

    void resize(const size_t &new_capacity);

    bool Empty();

    size_t Length();

    void Clear();

    ~TVector();

private:
    size_t capacity;
    size_t size;

    TVectorItem<T> *ptr;
};
#endif
```

tvector.cpp:

```
#include "tvector.h"

template<typename T>
TVector<T>::TVector() {
    capacity = 1;
    size = 0;
    ptr = (TVectorItem<T> *) (malloc(sizeof(TVectorItem<T>) * capacity));
}

template<typename T>
TVector<T>::TVector(size_t size) : size(size) {
    capacity = size;
    ptr = (TVectorItem<T> *) (malloc(sizeof(TVectorItem<T>) * capacity));
}

template<typename T>
TVector<T>::TVector(size_t size, T filler) : size(size) {
    capacity = size;
    ptr = (TVectorItem<T> *) (malloc(sizeof(TVectorItem<T>) * capacity));
    for (size_t i = 0; i < capacity; ++i) {
        new(ptr + i) TVectorItem<T>(filler);
    }
```

```
}

template<typename T>
TVector<T>::TVector(const TVector &other) {
    size = other.size;
    capacity = other.capacity;
    ptr = (TVectorItem<T> *) (malloc(sizeof(TVectorItem<T>) * capacity));
    for (size_t i = 0; i < size; ++i) {
        ptr[i] = other.ptr[i];
    }
}

template<typename T>
void TVector<T>::InsertLast(const T &elem) {
    if (size == capacity) {
        resize(capacity * 2);
    }
    new(ptr + size) TVectorItem<T>(elem);
    ++size;
}

template<typename T>
void TVector<T>::RemoveLast() {
    if (size == 0) {
        std::cout << "The length is zero!\n";
        return;
    }
    --size;
}

template<typename T>
const T &TVector<T>::Last() {
    return ptr[(size - 1)].GetValue();
}

template<typename T>
T &TVector<T>::operator[](const size_t &idx) {
    return ptr[idx].GetValue();
}

template<typename T>
bool TVector<T>::Empty() {
    return size == 0;
}

template<typename T>
size_t TVector<T>::Length() {
    return size;
}

template<typename T>
void TVector<T>::Clear() {
    size = 0;
}
```

```
template<typename T>
TVector<T>::~TVector<T>() {
    free(ptr);
}

template<typename T>
void TVector<T>::resize(const size_t &new_capacity) {
    capacity = new_capacity;
    ptr = (TVectorItem<T> *) realloc(ptr, capacity * sizeof(TVectorItem<T>));
}

template
class TVector<void *>;
```

tvector_item.h:

```
#ifndef TVECTOR_ITEM_H
#define TVECTOR_ITEM_H

#include "memory"
#include "new"

template<typename T>
class TVectorItem {
public:
    TVectorItem();

    TVectorItem(const TVectorItem<T> &other);

    TVectorItem(const T &item);

    TVectorItem<T> &operator=(const TVectorItem<T> &other);

    T &GetValue();

    virtual ~TVectorItem();

private:
    T value;
};


#endif
```

tvector_item.cpp:

```
#include "tvector_item.h"

template<typename T>
TVectorItem<T>::TVectorItem() {
    value = nullptr;
}

template<typename T>
TVectorItem<T>::TVectorItem(const TVectorItem<T> &other) {
    value = other.value;
}
```

```
template<typename T>
TVectorItem<T>::TVectorItem(const T &item) {
    value = item;
}

template<typename T>
T &TVectorItem<T>::GetValue() {
    return value;
}

template<typename T>
TVectorItem<T> &TVectorItem<T>::operator=(const TVectorItem<T> &other) {
    value = other.value;
    return *this;
}

template<typename T>
TVectorItem<T>::~TVectorItem<T>() {}

template
class TVectorItem<void *>;
```

tallocator.h:

```
#ifndef TALLOCATOR
#define TALLOCATOR

#include "tvector.h"
#include <new>

class TAllocator {
public:
    TAllocator(const size_t &block_size, const size_t &block_count);

    void *allocate(size_t size);

    void deallocate(void *ptr);

    bool hasFreeBlocks();

    ~TAllocator();

private:
    size_t free_count;
    size_t block_count;
    size_t block_size;

    TVector<void *> free_blocks;
    char *used_blocks;
};

#endif
```

tallocator.cpp:

```
#include "tallocator.h"
```

```
TAllocator::TAllocator(const size_t &block_size, const size_t &block_count) :
block_size(block_size),
                                                                    block_count(
block_count),
                                                                    free_count(b
lock_count) {
    used_blocks = (char *) malloc(block_size * block_count);
    for (size_t i = 0; i < block_count; ++i) {
        free_blocks.InsertLast(used_blocks + i * block_size);
    }
}

void *TAllocator::allocate(size_t size) {
    std::cout << "allocated!\n";
    if (!hasFreeBlocks()) {
        size_t old = block_count;
        free_count += 20;
        block_count += 20;
        used_blocks = (char *) realloc(used_blocks, block_count * block_size);
        for (size_t i = old; i < block_count; ++i) {
            free_blocks.InsertLast(used_blocks + i * block_size);
        }
    }
    void *result = free_blocks.Last();
    free_blocks.RemoveLast();
    --free_count;
    return result;
}

void TAllocator::deallocate(void *ptr) {
    std::cout << "deallocated!\n";
    free_count++;
    free_blocks.InsertLast(ptr);
}

bool TAllocator::hasFreeBlocks() {
    return free_count > 0;
}

TAllocator::~TAllocator() {
    std::cout << "freed!\n";
    free(used_blocks);
}
```

Результат работы:

```
Enter a Rhombus: 0 2 2 0 0 -2 -2 0
allocated!
Enter a Rhombus: 0 3 3 0 0 -3 -3 0
allocated!
Enter a Rhombus: 0 1 1 0 0 -1 -1 0
allocated!
```

```
Enter a Rhombus: 0 5 5 0 0 -5 -5 0
allocated!
created
Rhombus: (0, 2) (2, 0) (0, -2) (-2, 0)

Rhombus: (0, 3) (3, 0) (0, -3) (-3, 0)

Rhombus: (0, 1) (1, 0) (0, -1) (-1, 0)

Rhombus: (0, 5) (5, 0) (0, -5) (-5, 0)

deallocated!
deallocated!
deallocated!
deallocated!
freed!
```