**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ**

**(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСТИТЕТ)**

# ЛАБОРАТОРНАЯ РАБОТА №8

**по курсу объектно-ориентированное программирование I семестр, 2021/22 уч. год**

Студент *Алапанова Эльза Халилевна, группа М8О-207Б-20*

Преподаватель *Дорохов Евгений Павлович*

## Условие

Вариант 4:
Фигура: трапеция,
Контейнер: динамический массив (TVector).
Задание: Используя структуру данных, разработанную для лабораторной работы No5, спроекти- ровать и разработать аллокатор памяти для динамической структуры данных. Цель построения аллокатора – минимизация вызова операции malloc. Аллокатор дол- жен выделять большие блоки памяти для хранения фигур и при создании новых фигур- объектов выделять место под объекты в этой памяти. Алокатор должен хранить списки использованных/свободных блоков. Для хранения списка свободных блоков нужно применять динамическую структуру данных (контей- нер 2-го уровня, согласно варианту задания). Для вызова аллокатора должны быть переопределены оператор new и delete у классов-фигур.

Нельзя использовать:

1. Стандартные контейнеры std.

Программа должна позволять:

1. Вводить произвольное количество фигур и добавлять их в контейнер.
2. Распечатывать содержимое контейнера.
3. Удалять фигуры из контейнера.

Исходный код лежит в 11 файлах:

1. main.cpp: основная программа, взаимодействие с пользователем посредством команд из меню.
2. figure.h: описание абстрактного класса фигур.
3. point.h: описание класса точки.
4. tvector_item.h: описание класса элемента динамического массива.
5. tvector.h: описание класса динамического массива.
6. trapezoid.h: описание класса трапеции, наследующегося от figures.
7. point.cpp: реализация класса точки.
8. rectangle.cpp: реализация класса трапеции, наследующегося от figures.
9. tvector.cpp: реализация класса динамического массива.
10. tvector_item.cpp: реализация класса эелемента динамического массива.

11. titerator.h описание класса итератора.
12. TLinkedList.h
13. TLinkedList.cpp
14. TLinkedListItem.cpp
15. TLinkedListItem.h
16. tallocation_block.cpp: реализация аллокатора
17. tallocation_block.h: описание аллокатора

## Дневник отладки

Никаких проблем не возникало.

## Недочёты

Недочетов не заметила.

## Выводы

В данной лабораторной работе я реализовала аллокатор класса. Благодаря аллокаторам мы можем придумать свои правила для выделения памяти и снизить количество вызовов операции malloc.

**Исходный код:**

## figure.h

```
#ifndef FIGURE_H
#define FIGURE_H

class Figure
{
public:
        virtual size_t VertexesNumber() = 0;
        virtual double Area() = 0;
        virtual void Print() = 0;

        virtual ~Figure() {};
};

#endif /* FIGURE_H */
```

## point.h

```
#ifndef POINT_H
#define POINT_H

#include <iostream>

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

  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_H
```

## point.cpp

```
#include "point.h"
#include <cmath>
#include <iostream>

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_;
        is >> 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_;
        is >> p.y_;
        return is;
}

std::ostream& operator << (std::ostream& os, Point& p) {
        os << "(" << p.x_ << ", " << p.y_ << ")";
        return os;
}
```

## trapezoid.h

```cpp
#ifndef TRAPEZOID_H
#define TRAPEZOID_H

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

class Trapezoid : public Figure
{
public:
        Trapezoid();
        Trapezoid(Point a_, Point b_, Point c_, Point d_);
        Trapezoid(std::istream &is);
        Trapezoid(const Trapezoid& other);

        friend std::istream& operator>>(std::istream& is, Trapezoid& obj);
        friend std::ostream& operator<<(std::ostream& os, Trapezoid& obj);
        Trapezoid& operator=(const Trapezoid& right);
        size_t VertexesNumber();
        double Area();
        void Print();

        virtual ~Trapezoid(); // деструктор

private:
        Point a;
        Point b;
        Point c;
        Point d;
};

#endif /* TRAPEZOID_H */
```

## trapezoid.cpp

```cpp
#include <iostream>
#include <cmath>
#include <cstdint>
#include "trapezoid.h"

Trapezoid::Trapezoid() :
a(Point()), b(Point()), c(Point()), d(Point()) {
```

```
        std::cout << "Default constructor(Trapezoid)" << std::endl;
}

Trapezoid::Trapezoid(Point a_, Point b_, Point c_, Point d_) :
a(a_), b(b_), c(c_), d(d_) {
        std::cout << "Constructor2(Trapezoid): " << a_ << " " << b_ << " "
        << c_ << " " << d_ << std::endl;
}

Trapezoid::Trapezoid(std::istream &is) // istream потоки ввода вывода
{
        is >> a;
        is >> b;
        is >> c;
        is >> d;
        std::cout << "Constructor(Trapezoid) via istream : " << a << " " << b << " "
        << c << " " << d << std::endl;
}

Trapezoid::Trapezoid(const Trapezoid& other)
        : Trapezoid(other.a, other.b, other.c, other.d) {
        std::cout << "Trapezoid copy created" << std::endl;
}

std::istream& operator>>(std::istream& is, Trapezoid& obj) {
        std::cout << "Enter data: ";
        is >> obj.a >> obj.b >> obj.c >> obj.d;
        std::cout << "Trapezoid was created." << std::endl;
        return is;
}

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

Trapezoid& Trapezoid::operator=(const Trapezoid& right) {
        if (this == &right) {return *this; }
        a = right.a;
        b = right.b;
        c = right.c;
        d = right.d;
```

```
        return *this;
}

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

double Trapezoid::Area()
{
        double ax = a.X() - c.X();
        double bx = b.X() - d.X();
        double ay = a.Y() - c.Y();
        double by = b.Y() - d.Y();
        double COS = (ax * bx + ay * by) / (sqrt(ax * ax + ay * ay) *
                sqrt (bx * bx + by * by));
        return double(a.dist(c) * b.dist(d) * 0.5 * sin(acos(COS)));
}

void Trapezoid::Print()
{
        std::cout << "a = " << a << ", " << "b = "
        << b << ", " << "c = " << c << ", " << "d = " << d << std::endl;
}

Trapezoid::~Trapezoid()
{

}
```

# tvector_item.h

```
#ifndef TVECTORITEM_H
#define TVECTORITEM_H
#include <iostream>
#include "trapezoid.h"
#include "tallocation_block.h"
#include <memory>

template <class T> class TVectorItem {
public:
        TVectorItem(std::shared_ptr<T>& figure);
        template <class A> friend std::ostream& operator<<(std::ostream& os,
        const std::shared_ptr<TVectorItem<A>>& obj);
```

```
        std::shared_ptr<T> GetFigure() const;
        std::shared_ptr<TVectorItem<T>> GetNext();
        void SetNext(std::shared_ptr<TVectorItem<T>> next);
        void* operator new (size_t size);
        void operator delete(void* p);

        virtual ~TVectorItem();

private:
        std::shared_ptr<T> figure;
        std::shared_ptr<TVectorItem<T>> next;
        static TAllocationBlock vectoritem_allocator;
};

#endif
```

## tvector_item.cpp

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

template <class T> TVectorItem<T>::TVectorItem(std::shared_ptr<T>& figure) {
        this->figure = figure;
        this->next = nullptr;
        std::cout << "TVector Item: created" << std::endl;
}

template <class T> TAllocationBlock TVectorItem<T>::vectoritem_allocator(sizeof(TVectorI

template <class T> std::ostream& operator<<(std::ostream& os,
const std::shared_ptr<TVectorItem<T>>& obj) {
        obj->figure->Print();
        return os;
}

template <class T> std::shared_ptr<T> TVectorItem<T>::GetFigure() const {
        return this->figure;
}

template <class T> std::shared_ptr<TVectorItem<T>> TVectorItem<T>::GetNext() {
        this->next = next;
}
```

```
template <class T> void TVectorItem<T>::SetNext(std::shared_ptr<TVectorItem<T>> next) {
        this->next = next;
}

template <class T> void* TVectorItem<T>::operator new (size_t size) {
        return vectoritem_allocator.Allocate();
}

template <class T> void TVectorItem<T>::operator delete(void* p) {
        vectoritem_allocator.Deallocate(p);
}

template <class T> TVectorItem<T>::~TVectorItem() {

}

template class TVectorItem<Figure>;
template std::ostream& operator<<(std::ostream& os,
        const std::shared_ptr<TVectorItem<Figure>>& obj);
```

## tvector.h

```
#ifndef TVECTOR_H
#define TVECTOR_H

#include <iostream>
#include "tvector_item.h"
#include "trapezoid.h"
#include "titerator.h"
#include <memory>

template <class T> class TVector {
public:
    TVector();
    TVector(size_t size);
    bool Empty();

    size_t Length();
    void Resize(size_t new_size);
    void InsertLast(std::shared_ptr<T>& element);
    void Remove(int idx);
    void Print();
```

```
    void Print_idx(int idx);

    TIterator<TVectorItem<T>,T> begin();
    TIterator<TVectorItem<T>,T> end();

    virtual ~TVector();

private:
    size_t size;
    std::shared_ptr<TVectorItem<T>>* data;
};

#endif //TVECTOR_H
```

## tvector.cpp

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

template <class T> TVector<T>::TVector() {

}
template <class T> TVector<T>::TVector(size_t size)
{
        this->size = 0;
        this->data = (std::shared_ptr<TVectorItem<T>>*)malloc(sizeof
                (std::shared_ptr<TVectorItem<T>>) * size);
}

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

template <class T> size_t TVector<T>::Length()
{
        return this->size;
}

template <class T> void TVector<T>::Resize(size_t new_size) {
        this->data = (std::shared_ptr<TVectorItem<T>>*)realloc(this->data,
                sizeof(Trapezoid) * new_size);
}
```

```
template <class T> void TVector<T>::InsertLast(std::shared_ptr<T>& element) {
        this->data[size] = std::make_shared<TVectorItem<T>>(element);
        (this->size)++;
}

template <class T> void TVector<T>::Remove(int idx) {
        for (int j = idx; j != this->size - 1; j++) {
                this->data[j] = this->data[j + 1];
        }
        this->Resize(--(this->size));
}

template <class T> void TVector<T>::Print() {
        for(int i = 0; i < size; i++) {
                std::cout << data[i];
        }
}

template <class T> void TVector<T>::Print_idx(int idx) {
        std::cout << this->data[idx];
}

template <class T> TIterator<TVectorItem<T>,T> TVector<T>::begin() {
        return this->data[0];
}

template <class T> TIterator<TVectorItem<T>,T> TVector<T>::end() {
        return this->data[this->size];
}
template <class T> TVector<T>::~TVector()
{
        std::cout << "TVector was deleted" << std::endl;
}

template class TVector<Figure>;
```

## main.cpp

```
#include <iostream>
#include <cstdlib>
#include <memory>
#include "trapezoid.h"
#include "tvector.h"
```

```cpp
#include "tvector_item.h"
#include "titerator.h"

typedef unsigned long long Type;


int main()
{
    TVector<Figure>* v = new TVector<Figure>(1);
    std::shared_ptr<Figure> tmp;

    short int code = 0, number = 0;
    int index = 0;

    std::cout << "Code 1 means add element.\n" << std::endl;
    std::cout << "Code 2 means print element.\n" << std::endl;
    std::cout << "Code 3 means delete element.\n" << std::endl;
    std::cout << "Code 4 means break.\n" << std::endl;

    while (code != 4) {
        std::cout << "Code:" << std::endl;
        std::cin >> code;

        if (code == 1) {
            std::cout << "Write coordinates of Trapezoid" << std::endl;
            tmp = std::make_shared<Trapezoid>(std::cin);
            v->InsertLast(tmp);
            v->Resize((v->Length()) + 1);
        }

        if (code == 2) {
            std::cout << "You can print elements. Write it's index." << std::endl;
            std::cin >> index;
            if (index >= 0 && index < v->Length()) {
                v->Print_idx(index);
            } else {std::cout << "Incorrect index." << std::endl;}
        }

        if (code == 3) {
            std::cout << "You can delete elements. Write index." << std::endl;
            std::cin >> index;
            if (index >= 0 && index < v->Length()) {
```

```
                std::cout << "Size now: " << std::endl;
                std::cout << v->Length() << std::endl;
                v->Remove(index);
                std::cout << "Element was deleted, size of array was reduced."
                << std::endl;
                std::cout << "Size after: " << std::endl;
                std::cout << v->Length() << std::endl;
            }
            else {std::cout << "Incorrect index." << std::endl;}
        }
    }

    return 0;
}
```

## titerator.h

```
#ifndef TITERATOR_H
#define TITERATOR_H

#include <iostream>
#include <memory>

template <class node, class T>
class TIterator {
public:
    TIterator(std::shared_ptr<node> n) {
        node_ptr = n;
    }
    std::shared_ptr<T> operator*() {
        return node_ptr->GetValue();
    }
    std::shared_ptr<T> operator->() {
        return node_ptr->GetValue();
    }
    void operator++() {
        node_ptr = node_ptr->GetNext();
    }
    TIterator operator++(int) {
        TIterator iter(*this);
        ++(*this);
        return iter;
    }
```

```
    bool operator==(TIterator const& i) {
        return node_ptr == i.node_ptr;
    }
    bool operator!=(TIterator const& i) {
        return !(*this == i);
    }

private:
    std::shared_ptr<node> node_ptr;
};

#endif  // TITERATOR_H
```

# TLinkedList.h

```
#ifndef TLINKEDLIST_H
#define TLINKEDLIST_H

#include "TLinkedListItem.h"
#include <memory>
#include <iostream>

class TLinkedList {
public:
    TLinkedList();
    void InsertFirst(void *link);
    void InsertLast(void *link);
    void Insert(int position, void *link);
    int Length();
    bool Empty();
    void Remove(int &position);
    void Clear();

    void* GetBlock();

    virtual ~TLinkedList();
private:
    TLinkedListItem* first;
};

#endif // TLINKEDLIST_H
```

# TLinkedList.cpp

```
#include "TLinkedList.h"

TLinkedList::TLinkedList() {
    first = nullptr;
}

void TLinkedList::InsertFirst(void* link) {
    auto *other = new TLinkedListItem(link);
    other->SetNext(first);
    first = other;
}

void TLinkedList::Insert(int position, void *link) {
    TLinkedListItem *iter = this->first;
    auto *other = new TLinkedListItem(link);
    if (position == 1) {
        other->SetNext(iter);
        this->first = other;
    } else {
        if (position <= this->Length()) {
            for (int i = 1; i < position - 1; ++i)
                iter = iter->GetNext();
            other->SetNext(iter->GetNext());
            iter->SetNext(other);
        }
    }
}

void TLinkedList::InsertLast(void *link) {
    auto *other = new TLinkedListItem(link);
    TLinkedListItem *iter = this->first;
    if (first != nullptr) {
        while (iter->GetNext() != nullptr) {
            iter = iter->SetNext(iter->GetNext());
        }
        iter->SetNext(other);
        other->SetNext(nullptr);
    }
    else {
        first = other;
    }
```

```
}

int TLinkedList::Length() {
    int len = 0;
    TLinkedListItem* item = this->first;
    while (item != nullptr) {
        item = item->GetNext();
        len++;
    }
    return len;
}

bool TLinkedList::Empty() {
    return first == nullptr;
}

void TLinkedList::Remove(int &position) {
    TLinkedListItem *iter = this->first;
    if (position <= this->Length()) {
        if (position == 1) {
            this->first = iter->GetNext();
        } else {
            int i = 1;
            for (i = 1; i < position - 1; ++i) {
                iter = iter->GetNext();
            }
            iter->SetNext(iter->GetNext()->GetNext());
        }

    } else {
        std::cout << "error" << std::endl;
    }
}

void TLinkedList::Clear() {
    first = nullptr;
}

void * TLinkedList::GetBlock() {
    return this->first->GetBlock();
}
```

```
TLinkedList::~TLinkedList() {
    delete first;
}
```

# TLinkedListItem.h

```
#ifndef TLINKEDLISTITEM_H
#define TLINKEDLISTITEM_H

#include <memory>

class TLinkedListItem {
public:
    TLinkedListItem(void *link);
    void* GetBlock();

    TLinkedListItem* SetNext(TLinkedListItem* next);
    TLinkedListItem* GetNext();

    virtual ~TLinkedListItem();
private:
    void* link;
    TLinkedListItem* next;
};

#endif // TLINKEDLISTITEM_H
```

# TLinkedListItem.cpp

```cpp
#include "TLinkedListItem.h"
#include <iostream>

TLinkedListItem::TLinkedListItem(void* l) {
    this->link = l;
    this->next = nullptr;
}

TLinkedListItem* TLinkedListItem::SetNext(TLinkedListItem* n) {
    TLinkedListItem* o = this->next;
    this->next = n;
    return o;
}

TLinkedListItem* TLinkedListItem::GetNext() {
    return this->next;
}

void* TLinkedListItem::GetBlock() {
    return this->link;
}

TLinkedListItem::~TLinkedListItem() {
}
```