

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

## **ЛАБОРАТОРНАЯ РАБОТА №8**

по курсу “Объектно-ориентированное программирование”

I семестр, 2021/22 учебный год

Студентка: Ивченко Анна Владимировна

Группа: М8О-208Б-20

Преподаватель: Дорохов Евгений Павлович, каф. 806

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 №28:**

- Фигуры: Трапеция
- Контейнер первого уровня: Очередь
- Контейнер второго уровня: Связный список

**Описание программы:**

Программа состоит из 14 файлов:

- `main.cpp` – основная программа
- `figure.h` – описание класса фигуры
- `point.h` – описание класса точки
- `point.cpp` – реализация класса точки
- `trapezoid.h` – описание класса трапеции (наследуется от фигуры)
- `trapezoid.cpp` – реализация класса трапеции
- `TQueueItem.h` – описание элемента очереди
- `TQueueItem.cpp` – реализация элемента очереди
- `TQueue.h` – описание очереди
- `TQueue.cpp` – реализация очереди
- `TIterator.h` – реализация итератора
- `TAllocatorBlock.h/cpp` – реализация класса аллокатора для фигуры
- `TLinkedList.h/cpp` – реализация класса связного списка для использования в аллокаторе
- `TLinkedListItem.h/cpp` – реализация класса элемента связного списка для использования в аллокаторе

### **Дневник отладки:**

В ходе выполнения работы ошибок выявлено не было.

### **Тестирование:**

Default queue created

Queue: => =>

Enter n: 2

Enter points: 0 0 2 3 4 3 6 0

Trapezoid: (0, 0) (2, 3) (4, 3) (6, 0)

Allocated :40bytes

Queue item: created

Added one trapezoid to tail. Coordinates: Trapezoid: (0, 0) (2, 3) (4, 3) (6, 0)

. Area = 12

Queue: => 12 =>

Length: 1

Enter points: 6 6 7 7 8 7 9 6

Trapezoid: (6, 6) (7, 7) (8, 7) (9, 6)

Allocated :40bytes

Queue item: created

Added one trapezoid to tail. Coordinates: Trapezoid: (6, 6) (7, 7) (8, 7) (9, 6)

. Area = 2

Queue: => 2 12 =>

Length: 2

Queue copied

Queue: Queue: => 2 12 =>

Queue2: Queue: => 2 12 =>

Trapezoid: (0, 0) (2, 3) (4, 3) (6, 0)

Trapezoid: (6, 6) (7, 7) (8, 7) (9, 6)

### **Вывод:**

В ходе проделанной работы были освоены принципы реализации и работы с аллокаторами, или как их еще называют распределителем памяти. Я считаю, крайне важно уметь работать с памятью. Также я заметила, что аллокаторы достаточно редко встречаются в программировании и не думаю, что еще пригодится мне в дальнейшей работе.

### **Исходный код:**

#### **main.cpp:**

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

```

int main(int argc, char** argv) {
    TQueue<Trapezoid> queue;
    std::shared_ptr<Trapezoid> tr(new Trapezoid(1, 2, 3, 4));
    std::cout << queue << std::endl;
    std::shared_ptr<Trapezoid> t;
    std::cout << "Enter n: ";
    int n; std::cin >> n;
    for (int i = 0; i < n; i++) {
        std::cin >> *tr;
        std::cout << *tr << std::endl;
        queue.Push(std::shared_ptr<Trapezoid>(new Trapezoid(*tr)));
        std::cout << queue;
        std::cout << std::endl;
        std::cout << "Length: " << queue.Length() << std::endl;
    }
    TQueue<Trapezoid> queue2 = queue;
    std::cout << "Queue: " << queue << std::endl;
    std::cout << "Queue2: " << queue2 << std::endl;
    for (auto i : queue) {
        std::cout << *i << std::endl;
    }
    return 0;
}

```

### **figure.h:**

```

#ifndef FIGURE_H
#define FIGURE_H
#include <iostream>
class Figure {
public:
    virtual size_t VertexesNumber() = 0;
    virtual double Area() = 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);
    void SetX(double x);
    void SetY(double y);
    double GetX();
    double GetY();
    friend std::istream& operator>>(std::istream& is, Point& p);
    friend std::ostream& operator<<(std::ostream& os, Point& p);
    friend std::ostream& operator<<(std::ostream& os, const Point& p);
public:
    double x_;
    double y_;
};

```

```
#endif // POINT_H
```

### **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_;
}

void Point::SetX(double x) {
    this->x_ = x;
}
void Point::SetY(double y) {
    this->y_ = y;
}
double Point::GetX() {
    return this->x_;
}
double Point::GetY() {
    return this->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;
}
std::ostream& operator<<(std::ostream& os, const Point& p) {
    os << "(" << p.x_ << ", " << p.y_ << ")";
    return os;
}
```

### **trapezoid.h:**

```
#ifndef TRAPEZOID_H
#define TRAPEZOID_H
#include "figure.h"
#include <iostream>
#include "point.h"
#include <memory>
#include "TAllocationBlock.h"
class Trapezoid : public Figure {
public:
    Trapezoid();
    Trapezoid(double a, double b, double c, double d);
    Trapezoid(std::shared_ptr<Trapezoid>& other);
```

```

friend std::istream& operator>>(std::istream& is, Trapezoid& obj);
friend std::ostream& operator<<(std::ostream& os, const Trapezoid& obj);
Trapezoid& operator=(const Trapezoid& right);
bool operator==(const Trapezoid& right);
virtual ~Trapezoid();
size_t VertexesNumber();
double Area();
void *operator new(size_t size);
void operator delete(void *ptr);
public:
    double len_ab, len_bc, len_cd, len_da;
    Point a_, b_, c_, d_;

private:
    static TAllocationBlock block;
};
#endif // TRAPEZOID_H

```

## trapezoid.cpp:

```

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

Trapezoid::Trapezoid()
: len_ab(0.0),
  len_bc(0.0),
  len_cd(0.0),
  len_da(0.0) {
}

Trapezoid::Trapezoid(double ab, double bc, double cd, double da)
: len_ab(ab),
  len_bc(bc),
  len_cd(cd),
  len_da(da) {
}

Trapezoid::Trapezoid(std::shared_ptr<Trapezoid>& other)
: Trapezoid(other->len_ab, other->len_bc, other->len_cd, other->len_da) {
}

std::istream& operator>>(std::istream& is, Trapezoid& obj) {
    std::cout << "Enter points: ";
    is >> obj.a_;
    is >> obj.b_;
    is >> obj.c_;
    is >> obj.d_;
    obj.len_ab = obj.a_.dist(obj.b_);
    obj.len_bc = obj.b_.dist(obj.c_);
    obj.len_cd = obj.c_.dist(obj.d_);
    obj.len_da = obj.d_.dist(obj.a_);
    return is;
}

std::ostream& operator<<(std::ostream& os, const Trapezoid& obj) {
    std::cout << "Trapezoid: ";
    os << obj.a_; std::cout << " ";
    os << obj.b_; std::cout << " ";
    os << obj.c_; std::cout << " ";
    os << obj.d_; std::cout << std::endl;
    return os;
}

```

```

Trapezoid& Trapezoid::operator=(const Trapezoid& other) {
    if (this == &other)
        return *this;
    len_ab = other.len_ab;
    len_bc = other.len_bc;
    len_cd = other.len_cd;
    len_da = other.len_da;
    a_.x_ = other.a_.x_;
    a_.y_ = other.a_.y_;
    b_.x_ = other.b_.x_;
    b_.y_ = other.b_.y_;
    c_.x_ = other.c_.x_;
    c_.y_ = other.c_.y_;
    d_.x_ = other.d_.x_;
    d_.y_ = other.d_.y_;
    std::cout << "Trapezoid copied" << std::endl;
    return *this;
}

bool Trapezoid::operator==(const Trapezoid& other) {
    if (this->len_ab == other.len_ab &&
        this->len_bc == other.len_bc &&
        this->len_cd == other.len_cd &&
        this->len_da == other.len_da) {
        std::cout << "Trapezoids are equal" << std::endl;
        return 1;
    } else {
        std::cout << "Trapezoids are not equal" << std::endl;
        return 0;
    }
}

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

double Trapezoid::Area() {
    double p = (len_ab + len_bc + len_cd + len_da) / 2;
    return (len_bc + len_da) *
        std::sqrt((p - len_bc) *
            (p - len_da) *
            (p - len_da - len_ab) *
            (p - len_da - len_cd)) /
        std::abs(len_bc - len_da);
}

Trapezoid::~~Trapezoid() {
    std::cout << "Trapezoid deleted" << std::endl;
}

TAllocationBlock Trapezoid::block(sizeof(Trapezoid), 1000);

void *Trapezoid::operator new(size_t size) {
    return block.Allocate();
}

void Trapezoid::operator delete(void *ptr) {
    block.Deallocate(ptr);
}

```

## TQueueItem.h:

```

#ifndef TQUEUE_ITEM_H
#define TQUEUE_ITEM_H
#include <memory>

```

```

#include "trapezoid.h"
#include "TAllocationBlock.h"
template<typename T> class TQueueItem {
public:
    TQueueItem(const std::shared_ptr<T>& trapezoid);
    TQueueItem(const TQueueItem<T>& other);
    std::shared_ptr<TQueueItem<T>> SetNext(std::shared_ptr<TQueueItem> &next); ////////// added &
    std::shared_ptr<TQueueItem<T>> GetNext();

    std::shared_ptr<T> GetTrapezoid() const;
    template<typename A> friend std::ostream& operator<<(std::ostream& os, const TQueueItem<A>& obj);
    void* operator new(size_t size);
    void operator delete(void* p);
    virtual ~TQueueItem();
public:
    std::shared_ptr<T> item;
    std::shared_ptr<TQueueItem<T>> next;
};
#endif // TQUEUE_ITEM_H

```

## TQueueItem.cpp:

```

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

template <class T>
TQueueItem<T>::TQueueItem(const std::shared_ptr<T>& item) {
    this->item = item;
    this->next = nullptr;
    std::cout << "Queue item: created" << std::endl;
}

template <class T>
TQueueItem<T>::TQueueItem(const TQueueItem<T>& other) { // maybe change to TQueueItem<T>
    this->item = other.item;
    this->next = other.next;
    std::cout << "Queue item: copied" << std::endl;
}

template <class T>
std::shared_ptr<TQueueItem<T>> TQueueItem<T>::SetNext(
    std::shared_ptr<TQueueItem<T>> &next) { ////////// added &
    std::shared_ptr<TQueueItem<T>> old = this->next;
    this->next = next;
    return old;
}

template <class T>
std::shared_ptr<T> TQueueItem<T>::GetTrapezoid() const {
    return this->item;
}

template <class T>
std::shared_ptr<TQueueItem<T>> TQueueItem<T>::GetNext() {
    return this->next;
}

template <class T>
TQueueItem<T>::~TQueueItem() {
    std::cout << "Queue item: deleted" << std::endl;
}

```



```

template <class A>
std::ostream& operator<<(std::ostream& os, const TQueueItem<A>& obj) {
    os << obj.item->Area();
    return os;
}
template <class T>
void* TQueueItem<T>::operator new(size_t size) {
    std::cout << "Allocated :" << size << "bytes" << std::endl;
    return malloc(size);
}
template <class T>
void TQueueItem<T>::operator delete(void* p) {
    std::cout << "Deleted" << std::endl;
    free(p);
}
template class TQueueItem<Trapezoid>;
template std::ostream& operator<<(std::ostream& os, const TQueueItem<Trapezoid>& obj);

```

## TQueue.h:

```

#ifndef TQUEUE_H
#define TQUEUE_H
#include "tqueue_item.h"
#include "titerator.h"
#include <memory>
template <typename T> class TQueue {
public:
    TQueue();
    TQueue(const TQueue& other);
    void Push(std::shared_ptr<T> &&trapezoid);
    void Pop();
    std::shared_ptr<T>& Top();
    bool Empty();
    size_t Length();
    template <class A> friend std::ostream& operator<<(std::ostream& os, const TQueue<A>& queue);
    void Clear();
    TIterator<TQueueItem<T>, T> begin();
    TIterator<TQueueItem<T>, T> end();
    virtual ~TQueue();
private:
    std::shared_ptr<TQueueItem<T>> head, tail;
};
#endif // TQUEUE_H

```

## TQueue.cpp:

```

#include "tqueue.h"
#include <vector>
template <class T>
TQueue<T>::TQueue() : head(nullptr), tail(nullptr) {
    std::cout << "Default queue created" << std::endl;
}
template <class T>
TQueue<T>::TQueue(const TQueue& other) {
    head = other.head;
    tail = other.tail;
    std::cout << "Queue copied" << std::endl;
}

```

```

}
template <class T>
void TQueue<T>::Push(std::shared_ptr<T> &&trapezoid) {
    std::shared_ptr<TQueueItem<T>> other(new TQueueItem<T>(trapezoid));
    if (tail == nullptr) {
        head = tail = other;
        std::cout << "Added one trapezoid to tail. " << "Coordinates: " << *other->item << ". Area = " << other->item-
>Area() << std::endl;
        return;
    }
    tail->SetNext(other);
    tail = other;
    tail->next = nullptr;
    std::cout << "Added one trapezoid to tail. " << "Coordinates: " << *other->item << ". Area = " << other->item-
>Area() << std::endl;
}
template <class T>
void TQueue<T>::Pop() {
    if (head == nullptr)
        return;

    std::cout << "Removed one trapezoid from head." << "Coordinates: " << *head->item << ". Area = " << head-
>item->Area() << std::endl;
    head = head->GetNext();
    if (head == nullptr)
        tail = nullptr;
}
template <class T>
std::shared_ptr<T>& TQueue<T>::Top() {
    return head->item;
}
template <class T>
bool TQueue<T>::Empty() {
    return (head == nullptr) && (tail == nullptr);
}
template <class T>
size_t TQueue<T>::Length() {
    if (head == nullptr && tail == nullptr)
        return 0;
    std::shared_ptr<TQueueItem<T>> temp = head;
    int counter = 0;
    while (temp != tail->GetNext()) {
        temp = temp->GetNext();
        counter++;
    }
    return counter;
}
template <class T>
std::ostream& operator<<(std::ostream& os, const TQueue<T>& queue) {
    std::shared_ptr<TQueueItem<T>> temp = queue.head;
    std::vector<std::shared_ptr<TQueueItem<T>>> v;
    os << "Queue: ";
    os << "=> ";
    while (temp != nullptr) {
        v.push_back(temp);
        temp = temp->GetNext();
    }
    for (int i = v.size() - 1; i >= 0; --i)

```

```

        os << *v[i] << " ";
    os << "=>";
    return os;
}
template <class T>
void TQueue<T>::Clear() {
    for (int i = 0; i < this->Length(); i++) {
        this->Pop();
    }
    std::cout << "Queue was cleared but still exist" << std::endl;
}
template <class T>
TIterator<TQueueItem<T>, T> TQueue<T>::begin() {
    return TIterator<TQueueItem<T>, T>(head);
}
template <class T>
TIterator<TQueueItem<T>, T> TQueue<T>::end() {
    return TIterator<TQueueItem<T>, T>(nullptr);
}
template <class T>
TQueue<T>::~TQueue() {
    std::cout << "Queue was deleted" << std::endl;
}
template class TQueue<Trapezoid>;
template std::ostream& operator<<(std::ostream& os, const TQueue<Trapezoid>& queue);

```

## TIterator.cpp:

```

#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->GetTrapezoid(); }
    std::shared_ptr<T> operator->() { return node_ptr->GetTrapezoid(); }
    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

```

## TAllocationBlock.h:

```

#ifndef LAB6_TALLOCATIONBLOCK_H
#define LAB6_TALLOCATIONBLOCK_H
#include <iostream>
#include <cstdlib>

```

```

#include "TLinkedList.h"
class TAllocationBlock {
public:
    TAllocationBlock(int32_t size, int32_t count);
    void *Allocate();
    void Deallocate(void *ptr);
    bool Empty();
    int32_t Size();
    virtual ~TAllocationBlock();
private:
    char *used_bl;
    TLinkedList free_bl;
};
#endif //LAB6_TALLOCATIONBLOCK_H

```

## **TAllocationBlock.cpp:**

```

#include "TAllocationBlock.h"
TAllocationBlock::TAllocationBlock(int32_t size, int32_t count) {
    used_bl = (char *)malloc(size * count);
    for (int32_t i = 0; i < count; ++i) {
        void *ptr = (void *)malloc(sizeof(void *));
        ptr = used_bl + i * size;
        free_bl.InsertLast(ptr);
    }
}

void *TAllocationBlock::Allocate() {
    if (!free_bl.Empty()) {
        void *res = free_bl.GetBlock();
        int32_t first = 1;
        free_bl.Remove(first);
        std::cout << "Rectangle created" << std::endl;
        return res;
    } else {
        throw std::bad_alloc();
    }
}

void TAllocationBlock::Deallocate(void *ptr) {
    free_bl.InsertFirst(ptr);
}

bool TAllocationBlock::Empty() {
    return free_bl.Empty();
}

int32_t TAllocationBlock::Size() {
    return free_bl.Length();
}

TAllocationBlock::~TAllocationBlock() {
    while (!free_bl.Empty()) {
        int32_t first = 1;
        free_bl.Remove(first);
    }
    free(used_bl);
    std::cout << "Rectangle deleted" << std::endl;
}

```

## **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:**

```

#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() {
}

```

## **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;
}

```