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

Студент: *Мерц Савелий Павлович, группа М8О-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 №14:

- Фигура: Восьмиугольник (Octagon)
- Контейнер: Динамический массив (TVector)
- 2 контейнер: Связанный список (TLinkedList)

Описание программы:

Исходный код разделен на 10 файлов:

- point.hpp – описание класса точки
- point.cpp – реализация класса точки
- figure.hpp – описание класса фигуры
- octagon.hpp – описание класса восьмиугольника (наследуется от фигуры)
- octagon.cpp – реализация класса восьмиугольника
- tvector.hpp – реализация класса динамического массива
- iterator.hpp – реализация итератора
- main.cpp – основная программа
- TLinkedList.hpp - реализация списка
- TLinkedList_item.hpp - реализация элемента списка
- TAllocatorBlock.hpp - реализация аллокатора

Дневник отладки:

Проблем не возникало

Вывод:

В процессе выполнения работы я на практике познакомился с понятием аллокатора. Так как во многих структурах данных используются аллокаторы, то это очень важная тема, которую должен знать каждый программист на C++. Написание собственноручного итератора помогает реализовать собственную логику выделения памяти, которая может быть более оправданной в некоторых ситуациях, чем стандартный аллокатор, как для самописных, так и для стандартных структур данных.

Исходный код:

point.hpp:

```
#ifndef POINT_H
#define POINT_H

#include <iostream>

class Point {
public:
  Point();
  Point(double x, double y);

  double getX() const;
  double getY() const;

  friend std::istream& operator>>(std::istream& is, Point& p);
  friend std::ostream& operator<<(std::ostream& os, const Point& p);
  const Point& operator=(const Point& other);
  bool operator!=(const Point& other) const;
  bool operator==(const Point& other) const;
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) {}

double Point::getX() const{
	return x_;
}
```

```
double Point::getY() const{
	return y_;
}

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

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

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

bool Point::operator!=(const Point& other) const{
	if (x_ == other.x_) return false;
	if (y_ == other.y_) return false;
	return true;
}
bool Point::operator==(const Point& other) const{
	if (x_ != other.x_) return false;
	if (y_ != other.y_) return false;
	return true;
}
```

figure.hpp:

```
#ifndef FIGURE_H
#define FIGURE_H

#include <iostream>

class Figure {
public:
    size_t VertexesNumber();
    double Area();
    ~Figure() {};
};

#endif
```

octagon.hpp:

```
#ifndef OCTAGON_H
#define OCTAGON_H

#include <iostream>

#include "figure.h"
#include "point.h"
```

```
class Octagon : public Figure {
public:
    Octagon();
     Octagon(Point a, Point b, Point c, Point d, Point e, Point f, Point g, Point
h);

    friend std::istream& operator>>(std::istream& is,  Octagon& obj);
    friend std::ostream& operator<<(std::ostream& os, const Octagon& obj);

    size_t VertexesNumber();
    double Area() const;
    const Octagon& operator=(const Octagon& other);
    bool operator==(const Octagon& other) const;
    bool operator!=(const Octagon& other) const;

    ~Octagon();
private:
    Point a_, b_, c_, d_, e_, f_, g_, h_;
};

#endif
```

octagon.cpp:

```
#include "octagon.h"

#include <cmath>

Octagon::Octagon()
    : a_(0, 0),
     b_(0, 0),
     c_(0, 0),
     d_(0, 0),
     e_(0, 0),
     f_(0, 0),
     g_(0, 0),
     h_(0, 0) {
}

Octagon::Octagon(Point a, Point b, Point c, Point d, Point e, Point f, Point g,
Point h)
    : a_(a),
     b_(b),
     c_(c),
     d_(d),
     e_(e),
     f_(f),
     g_(g),
     h_(h) {
}

std::istream& operator>>(std::istream& is, Octagon& obj) {
    is >> obj.a_;
       is >> obj.b_;
       is >> obj.c_;
       is >> obj.d_;
       is >> obj.e_;
       is >> obj.f_;
       is >> obj.g_;
```

```
	is >> obj.h_;
	return is;
}

size_t Octagon::VertexesNumber() {
    return 8;
}

std::ostream& operator<<(std::ostream& os, const Octagon& obj) {
    os << "Octagon: ";
    os <<  obj.a_ << " ";
    os <<  obj.b_ << " ";
    os <<  obj.c_ << " ";
    os <<  obj.d_ << " ";
    os <<  obj.e_ << " ";
    os <<  obj.f_ << " ";
    os <<  obj.g_ << " ";
    os <<  obj.h_ << " ";
    return os;
}

double Octagon::Area() const{
	return   0.5   *   abs(   a_.getX()*b_.getY()   +   b_.getX()*c_.getY()   +
c_.getX()*d_.getY()    +    d_.getX()*e_.getY()    +    e_.getX()*f_.getY()    +
f_.getX()*g_.getY() + g_.getX()*h_.getY() + h_.getX()*a_.getY()
	-   a_.getY()*b_.getX()   -   b_.getY()*c_.getX()   -   c_.getY()*d_.getX()   -
d_.getY()*e_.getX()    -    e_.getY()*f_.getX()    -    f_.getY()*g_.getX()    -
g_.getY()*h_.getX() - h_.getY()*a_.getX());
}

const Octagon& Octagon::operator=(const Octagon& other) {
	if (this == &other)
	return *this;

	a_ = other.a_;
	b_ = other.b_;
	c_ = other.c_;
	d_ = other.d_;
	e_ = other.e_;
	f_ = other.f_;
	g_ = other.g_;
	h_ = other.h_;

	return *this;
}

bool Octagon::operator==(const Octagon& other) const{
    if (a_ != other.a_) return false;
    if (b_ != other.b_) return false;
    if (c_ != other.c_) return false;
    if (d_ != other.d_) return false;
    if (e_ != other.e_) return false;
    if (f_ != other.f_) return false;
    if (g_ != other.g_) return false;
    if (h_ != other.h_) return false;

    return true;
}
bool Octagon::operator!=(const Octagon& other) const{
```

```
    if (a_ != other.a_) return true;
    if (b_ != other.b_) return true;
    if (c_ != other.c_) return true;
    if (d_ != other.d_) return true;
    if (e_ != other.e_) return true;
    if (f_ != other.f_) return true;
    if (g_ != other.g_) return true;
    if (h_ != other.h_) return true;

    return false;
}

Octagon::~Octagon() {
}
```

TVector.hpp

```
#ifndef TVECTOR_H
#define TVECTOR_H

#include <iostream>
#include "iterator.hpp"
#include <memory>
#define SPTR(T) std::shared_ptr<T>

template <class Polygon>
class TVector
{
public:
    // Конструктор по умолчанию
    TVector();
    // изменение размера массива
    void Resize(size_t nsize);
    // Конструктор копирования
    TVector(const TVector& other);
    // Метод, добавляющий фигуру в конец массива
    void InsertLast(const Polygon& polygon);
    // Метод, удаляющий последнюю фигуру массива
    void RemoveLast();
    // Метод, возвращающий последнюю фигуру массива
    const Polygon& Last();
    // Перегруженный оператор обращения к массиву по индексу
    const SPTR(Polygon) operator[] (const size_t idx);
    // Метод, проверяющий пустоту
    bool Empty();
    // Метод, возвращающий длину массива
    size_t Length();
    // Оператор вывода для массива в формате:
    // "[S1 S2 ... Sn]", где Si - площадь фигуры
    template <class T>
    friend std::ostream& operator<<(std::ostream& os, const TVector<T>& arr);
    // Метод, удаляющий все элементы контейнера,
    // но позволяющий пользоваться им.
    void Clear();
    // Итератор начала
    Iterator<Polygon> begin(){
        return Iterator<Polygon>(data);
    }
    // Итератор конца
```

```
    Iterator<Polygon> end(){
        return Iterator<Polygon>(data + size);
    }

    void * operator new (size_t size);
    void operator delete(void *p);

    // Деструктор
    virtual ~TVector();
private:
    int size;
    SPTR(Polygon)* data;
};

#endif


template <class Polygon>
TVector<Polygon>::TVector(){
    size = 1;
    data = new SPTR(Polygon)[size];
}

template <class Polygon>
void TVector<Polygon>::Resize(size_t nsize){
      if(nsize == size)
        return;
    else{
        SPTR(Polygon)* ndata = new SPTR(Polygon)[nsize];
        for (int i = 0; i < (size < nsize ? size : nsize); i++)
            ndata[i] = data[i];
        delete[] data;
        data = ndata;
        size = nsize;
    }
}

template <class Polygon>
TVector<Polygon>::TVector(const TVector& other){
      size = other.size;
    data = new SPTR(Polygon)[other.size];
    for (int i = 0; i < size; i++)
        data[i] = other.data[i];
}

template <class Polygon>
void TVector<Polygon>::InsertLast(const Polygon& polygon){
      if (data[size - 1] != nullptr)
        Resize(size+1);
    data[size - 1] = std::make_shared<Polygon>(polygon);
}

template <class Polygon>
void TVector<Polygon>::RemoveLast(){
      data[size-1]=nullptr;
}

template <class Polygon>
const Polygon& TVector<Polygon>::Last(){
```

```
      return *(data[size - 1]);
}

template <class Polygon>
const SPTR(Polygon) TVector<Polygon>::operator[] (const size_t idx){
    if (idx >= 0 && idx < size)
        return data[idx];
    exit(1);
}

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

template <class Polygon>
size_t TVector<Polygon>::Length(){
      return size;
}

template <class Polygon>
std::ostream& operator<<(std::ostream& os, const TVector<Polygon>& arr){
    os << '[';
      for (size_t i = 0; i < arr.size; i++)
        os << (arr.data[i])->Area() << ((i != arr.size-1) ? ' ' : '\0');
    os << ']';
    return os;
}

template <class Polygon>
void TVector<Polygon>::Clear(){
      delete[] data;
    size = 1;
    data = new SPTR(Polygon)[size];
}

template <class Polygon>
void * TVector<Polygon>::operator new(size_t size)
{
    return queueItemAllocator.Allocate();
}

template <class Polygon>
void TVector<Polygon>::operator delete(void * p)
{
    queueItemAllocator.Deallocate(p);
}


template <class Polygon>
TVector<Polygon>::~TVector(){
    delete[] data;
}
```

Iterator.h

```
#ifndef ITERATOR_H
#define ITERATOR_H

#include <iostream>
```

```
#include <memory>

template <class Poligon>
class Iterator {
public:
    Iterator(std::shared_ptr<Poligon>* n){
        iter = n;
    }
    Poligon operator*(){
        return *(*iter);
    }
    Poligon operator->(){
        return *(*iter);
    }
    void operator++(){
        iter += 1;
    }
    Iterator operator++(int){
        Iterator iter(*this);
        ++(*this);
        return iter;
    }
    bool operator==(Iterator const& i) const{
        return iter == i.iter;
    }
    bool operator!=(Iterator const& i) const{
        return iter != i.iter;
    }

private:
    std::shared_ptr<Poligon>* iter;
};

#endif
```

main.cpp

```
#include <iostream>
#include "octagon.hpp"
#include "tvector.hpp"

int main()
{
    Octagon octi[8];
    TVector<Octagon> vec;
    for (int i = 0; i < 8; i++)
    {
        std::cin>>octi[i];
        vec.InsertLast(octi[i]);
    }
    for (auto i : vec) {
        std::cout << i << std::endl;
    }

    return 0;
}
```

TLinkedList.hpp

```
#ifndef TLINKEDLIST_H
#define TLINKEDLIST_H

#include "tlinkedlist_item.hpp"

template <class T>
class TLinkedList
{
public:
    TLinkedList() = default;
    TLinkedList(const TLinkedList<T> &other);

    T First() const;
    T Last() const;
    T GetItem(size_t position) const;

    void InsertFirst(const T item);
    void InsertLast(const T item);
    void Insert(const T item, size_t position);

    void RemoveFirst();
    void RemoveLast();
    void Remove(size_t position);

    bool Empty() const;
    size_t Length() const;

    TLinkedList<T>& operator=(const TLinkedList<T> &other);
    bool operator==(const TLinkedList<T> &other) const;
    bool operator!=(const TLinkedList<T> &other) const;

    // prints "S1 -> S2 -> ... -> Sn", S - area of square
    template <class A>
    friend std::ostream &operator<<(std::ostream &os, const TLinkedList<A> &l);

    void Clear();
    virtual ~TLinkedList();

private:
    std::shared_ptr<TLinkedListItem<T>> first = nullptr;
    std::shared_ptr<TLinkedListItem<T>> last = nullptr;
};

#endif // TLINKEDLIST_H

template <class T>
TLinkedList<T>::TLinkedList(const TLinkedList &o) : first(o.first), last(o.last)
{}

template <class T>
T TLinkedList<T>::First() const
{
    if (first == nullptr) {
        std::cout << "List is empty" << std::endl;
        T out(new T);
        return out;
    }
```

```
    return first->GetObject();
}

template <class T>
T TLinkedList<T>::Last() const
{
    if (last == nullptr) {
        std::cout << "List is empty" << std::endl;
        T out(new T);
        return out;
    }
    return last->GetObject();
}

template <class T>
T TLinkedList<T>::GetItem(size_t pos) const
{
    --pos;
    size_t n = this->Length();
    if (first == nullptr) {
        std::cout << "Err: List is empty" << std::endl;
        return nullptr;
    }
    if (pos > n - 1) {
        if (n == 1) {
            std::cout << "Err: pos = " << pos + 1 << "\n";
            std::cout << "Hint: Available pos is 1" << std::endl;
            return nullptr;
        }
        std::cout << "Err: pos = " << pos + 1 << "\n";
        std::cout << "Hint: Available pos is between 1 and " << n << std::endl;
        return nullptr;
    }
    if (pos == 0) {
        return first->GetObject();
    }
    if (pos == n - 1) {
        return last->GetObject();
    }
    std::shared_ptr<TLinkedListItem<T>> to_return = first;
    for (size_t i = 0; i < pos; ++i) {
        to_return = to_return->GetNext();
    }
    return to_return->GetObject();
}

template <class T>
void TLinkedList<T>::InsertFirst(T item)
{
    std::shared_ptr<TLinkedListItem<T>> new_item(new TLinkedListItem<T>(item));
    if (first == nullptr) {
        first = (last = new_item);
        return;
    }
    new_item->SetPrev(nullptr);
    new_item->SetNext(first);
    first->SetPrev(new_item);
    first = new_item;
}
```

```
template <class T>
void TLinkedList<T>::InsertLast(T item)
{
    std::shared_ptr<TLinkedListItem<T>> new_item(new TLinkedListItem<T>(item));
    if (first == nullptr) {
        first = (last = new_item);
        return;
    }
    new_item->SetPrev(last);
    new_item->SetNext(nullptr);
    last->SetNext(new_item);
    last = new_item;
}

template <class T>
void TLinkedList<T>::Insert(T item, size_t pos)
{
    --pos;
    size_t n = this->Length();
    if (pos > n) {
        if (n == 0) {
            std::cout << "Err: pos = " << pos + 1 << "\n";
            std::cout << "Hint: Available pos is 1" << std::endl;
            return;
        }
        std::cout << "Err: pos = " << pos + 1 << "\n";
            std::cout << "Hint: Available pos is between 1 and " << n + 1 <<
std::endl;
        return;
    }
    if (pos == 0) {
        InsertFirst(item);
        return;
    }
    if (pos == n) {
        InsertLast(item);
        return;
    }
    std::shared_ptr<TLinkedListItem<T>> new_item(new TLinkedListItem<T>(item));
    std::shared_ptr<TLinkedListItem<T>> cur = first;
    for (size_t i = 0; i < pos; ++i) {
        cur = cur->GetNext();
    }
    std::shared_ptr<TLinkedListItem<T>> cur_prev = cur->GetPrev();
    cur_prev->SetNext(new_item);
    cur->SetPrev(new_item);
    new_item->SetPrev(cur_prev);
    new_item->SetNext(cur);
}

template <class T>
void TLinkedList<T>::RemoveFirst()
{
    if (first == nullptr) {
        std::cout << "Err: List is empty" << std::endl;
        return;
    }
    if (last == first) {
```

```
        first = (last = nullptr);
        return;
    }
    std::shared_ptr<TLinkedListItem<T>> to_del = first;
    first = first->GetNext();
    first->SetPrev(nullptr);
}

template <class T>
void TLinkedList<T>::RemoveLast()
{
    if (last == nullptr) {
        std::cout << "Err: List is empty" << std::endl;
        return;
    }
    if (last == first) {
        first = (last = nullptr);
        return;
    }
    std::shared_ptr<TLinkedListItem<T>> to_del = last;
    last = last->GetPrev();
    last->SetNext(nullptr);
}

template <class T>
void TLinkedList<T>::Remove(size_t pos)
{
    --pos;
    size_t n = this->Length();
    if (first == nullptr) {
        std::cout << "Err: List is empty" << std::endl;
        return;
    }
    if (pos > n - 1) {
        if (n == 1) {
            std::cout << "Err: pos = " << pos + 1 << "\n";
            std::cout << "Hint: Available pos is 1" << std::endl;
            return;
        }
        std::cout << "Err: pos = " << pos + 1 << "\n";
        std::cout << "Hint: Available pos is between 1 and " << n << std::endl;
        return;
    }
    if (pos == 0) {
        RemoveFirst();
        return;
    }
    if (pos == n - 1) {
        RemoveLast();
        return;
    }
    std::shared_ptr<TLinkedListItem<T>> to_del = first;
    for (size_t i = 0; i < pos; ++i) {
        to_del = to_del->GetNext();
    }
    std::shared_ptr<TLinkedListItem<T>> cur_prev = to_del->GetPrev();
    std::shared_ptr<TLinkedListItem<T>> cur_next = to_del->GetNext();
    cur_prev->SetNext(cur_next);
    cur_next->SetPrev(cur_prev);
```

```
}

template <class T>
bool TLinkedList<T>::Empty() const
{
    return (first == nullptr);
}

template <class T>
size_t TLinkedList<T>::Length() const
{
    size_t res = 0;
       for (std::shared_ptr<TLinkedListItem<T>> i = first; i != nullptr; i =
i->GetNext()) {
        ++res;
    }
    return res;
}

template <class T>
TLinkedList<T>& TLinkedList<T>::operator=(const TLinkedList<T> &o)
{
    Clear();
      for (std::shared_ptr<TLinkedListItem<T>> i = o.first; i != nullptr; i =
i->GetNext()) {
        InsertLast(i->GetObject());
    }
    return *this;
}

template <class T>
bool TLinkedList<T>::operator==(const TLinkedList<T> &o) const
{
    if (Length() != o.Length()) {
        return false;
    }
    std::shared_ptr<TLinkedListItem<T>> i = first;
    std::shared_ptr<TLinkedListItem<T>> j = o.first;
    while (i != nullptr && j != nullptr) {
        if (i->GetObject() != j->GetObject()) {
            return false;
        }
        i = i->GetNext();
        j = j->GetNext();
    }
    return true;
}

template <class T>
bool TLinkedList<T>::operator!=(const TLinkedList<T> &o) const
{
    if (Length() != o.Length()) {
        return true;
    }
    std::shared_ptr<TLinkedListItem<T>> i = first;
    std::shared_ptr<TLinkedListItem<T>> j = o.first;
    while (i != nullptr && j != nullptr) {
        if (i->GetObject() != j->GetObject()) {
            return true;
```

```
        }
        i = i->GetNext();
        j = j->GetNext();
    }
    return false;
}

// prints "S1 -> S2 -> ... -> Sn", S - area of square
template <class T>
std::ostream &operator<<(std::ostream &os, const TLinkedList<T> &l)
{
    if (l.first == nullptr) {
        os << "List is empty" << std::endl;
        return os;
    }
      for (std::shared_ptr<TLinkedListItem<T>> i = l.first; i != nullptr; i =
i->GetNext()) {
        if (i->GetNext() != nullptr) {
            os << i->GetObject()->Area() << " -> ";
        } else {
            os << i->GetObject()->Area();
        }
    }
    return os;
}

template <class T>
void TLinkedList<T>::Clear()
{
    while (first != nullptr) {
        RemoveFirst();
    }
}

template <class T>
TLinkedList<T>::~TLinkedList()
{
    while (first != nullptr) {
        RemoveFirst();
    }
}
```

TLinkedList_item.hpp

```
#ifndef TLINKEDLIST_ITEM_H
#define TLINKEDLIST_ITEM_H

#include <iostream>
#include <memory>

template <class T>
class TLinkedListItem
{
public:
    TLinkedListItem(const std::shared_ptr<T> item);
    TLinkedListItem(const TLinkedListItem<T> &other);

    std::shared_ptr<TLinkedListItem<T>> GetPrev() const;
```

```
    std::shared_ptr<TLinkedListItem<T>> GetNext() const;
    std::shared_ptr<T> GetObject() const;

    void SetPrev(std::shared_ptr<TLinkedListItem<T>> to_set);
    void SetNext(std::shared_ptr<TLinkedListItem<T>> to_set);

    bool operator==(const TLinkedListItem<T> &other) const;
    bool operator!=(const TLinkedListItem<T> &other) const;

    template <class A>
      friend std::ostream& operator<<(std::ostream &os, const TLinkedListItem<A>
&i);

    ~TLinkedListItem();

private:
    std::shared_ptr<T> item;
    std::shared_ptr<TLinkedListItem<T>> prev;
    std::shared_ptr<TLinkedListItem<T>> next;
};

#endif // TLINKEDLIST_ITEM_H

template <class T>
TLinkedListItem<T>::TLinkedListItem(const std::shared_ptr<T> item) :
    item(item), prev(nullptr), next(nullptr) {}

template <class T>
TLinkedListItem<T>::TLinkedListItem(const TLinkedListItem<T> &o) :
    item(o.item), prev(o.prev), next(o.next) {}

template <class T>
std::shared_ptr<TLinkedListItem<T>> TLinkedListItem<T>::GetPrev() const
{
    return prev;
}

template <class T>
std::shared_ptr<TLinkedListItem<T>> TLinkedListItem<T>::GetNext() const
{
    return next;
}

template <class T>
std::shared_ptr<T> TLinkedListItem<T>::GetObject() const
{
    return item;
}

template <class T>
void TLinkedListItem<T>::SetPrev(std::shared_ptr<TLinkedListItem<T>> to_set)
{
    prev = to_set;
}

template <class T>
void TLinkedListItem<T>::SetNext(std::shared_ptr<TLinkedListItem<T>> to_set)
{
    next = to_set;
```

```
}

template <class T>
bool TLinkedListItem<T>::operator==(const TLinkedListItem<T> &o) const
{
    return ((item == o.item) && (prev == o.prev) && (next == o.next));
}

template <class T>
bool TLinkedListItem<T>::operator!=(const TLinkedListItem<T> &o) const
{
    return ((item != o.item) || (prev != o.prev) || (next != o.next));
}

template <class T>
std::ostream &operator<<(std::ostream &os, const TLinkedListItem<T> &i)
{
    os << "Item: " << *i.item << std::endl;
    return os;
}

template <class T>
TLinkedListItem<T>::~TLinkedListItem() {}
```

TAllocateBlock.hpp

```
#ifndef TALLOCATORBLOCK_H

#define TALLOCATORBLOCK_H



#include "TLinkedList.hpp"

#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.InsertLast(malloc(size));

            }

        }

        void* Allocate(const size_t& size){

            if(size != this->size){

                std::cout << "Error\n";
```

```
        }
        if(unused_blocks.Length()){
            for(int i = 0; i < 5; ++i){
                unused_blocks.InsertLast(malloc(size));
            }
        }
        void* tmp = unused_blocks.GetItem(1);
        used_blocks.InsertLast(unused_blocks.GetItem(1));
        unused_blocks.Remove(0);
        return tmp;
    }
    void Deallocate(void* ptr){
        unused_blocks.InsertLast(ptr);
    }
~TAllocatorBlock(){
    while(used_blocks.Length()){
        try{
            free(used_blocks.GetItem(1));
            used_blocks.Remove(0);
        } catch(...){
            used_blocks.Remove(0);
        }
    }
    while(unused_blocks.Length()){
        try{
            free(unused_blocks.GetItem(1));
            unused_blocks.Remove(0);
        } catch(...){
            unused_blocks.Remove(0);
```

```
            }
        }
    }

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

#endif
```