**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 *Белоусов Егор Владимирович, группа М8О-207Б-20\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

Преподаватель *Дорохов Евгений Павлович\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

### **Цель работы**

Целью лабораторной работы является:

* Закрепление навыков по работе с памятью в C++;
* Создание аллокаторов памяти для динамических структур данных.

### **Задание**

Используя структуру данных, разработанную для лабораторной работы №5, спроектировать и разработать аллокатор памяти для динамической структуры данных.

Цель построения аллокатора – 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 (динамический массив).

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

Нельзя использовать:

* Стандартные контейнеры std.

Программа должна позволять:

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

**Описание программы**  
Исходный код лежит в 16 файлах:

1. main.cpp основная программа

2. hexagon.h описание класса шестиугольник

3. hexagon.cpp реализация класса шестиугольник

4. point.h описание класса точка

5. point.cpp реализация класса точка

6. tlinkedlist\_item.h описание класса-элемента списка

7. tlinkedlist\_item.cpp реализация класса-элемента списка

8. tlinkedlist.h описание класса списка

9. tlinkedlist.cpp реализация класса списка

10. titerator.h описание и реализация класса итератора

11. tvector\_item.h описание класса элемента динамического массива

12. tvector\_item.cpp реализация класса элемента динамического массива

13. tvector.h описания класса динамического массива

14. tvector.cpp реализация класса динамического массива

15. tallocator.h описание класса аллокатора

16. tallocator.cpp реализация класса аллокатора

**Дневник отладки**

*D:\oop\lab6\cmake-build-debug\lab6.exe*

*Enter a Hexagon: 1 2 5 6 10 12 32 10 42 44*

*10 20*

*allocated!*

*Enter a Hexagon:20 30 11 120 32 10 22 50 60 10 21 23*

*allocated!*

*Enter a Hexagon:100 200 300 400 500 600 1 2 3 4 5 6*

*allocated!*

*Enter a Hexagon:12 11 10 9 8 7 6 5 4 3 2 1*

*allocated!*

*created!*

*Hexagon: (1, 2) (5, 6) (10, 12) (32, 10) (42, 44) (10, 20)*

*Hexagon: (20, 30) (11, 120) (32, 10) (22, 50) (60, 10) (21, 23)*

*Hexagon: (100, 200) (300, 400) (500, 600) (1, 2) (3, 4) (5, 6)*

*Hexagon: (12, 11) (10, 9) (8, 7) (6, 5) (4, 3) (2, 1)*

*deallocated!*

*deallocated!*

*deallocated!*

*deallocated!*

*freed!*

*Process finished with exit code 0*

**Недочёты**

По моему мнению недочетов нет.

**Выводы**

В данной лабораторной работе я познакомился с таким понятием как аллокатор. Создал собственный аллокатор для хранения фигур, а также реализовал вспомогательную структуру данных (динамический массив) для аллокатора.

**Исходный код**

main.cpp

#include "tlinkedlist.h"  
  
int main() {  
 Hexagon \*hex[4];  
  
 for (int i = 0; i < 4; ++i) {  
 std::cout << "Enter a Hexagon: ";  
 Hexagon hexagon;  
 std::cin >> hexagon;  
 hex[i] = new Hexagon(hexagon);  
 }  
 std::cout << "created!\n";  
  
 for (int i = 0; i < 4; ++i) {  
 std::cout << \*hex[i] << "\n";  
 }  
  
 for (int i = 0; i < 4; ++i) {  
 delete hex[i];  
 }  
 return 0;  
}

hexagon.h

#ifndef LAB6\_HEXAGON\_H  
#define LAB6\_HEXAGON\_H  
  
#include "iostream"  
#include "point.h"  
#include "tallocator.h"  
  
class Hexagon {  
private:  
 static const size\_t size = 6;  
 static TAllocator alloc;  
 Point P[size];  
public:  
 Hexagon();  
  
 double Area();  
  
 size\_t VertexesNumber();  
  
 Hexagon &operator=(const Hexagon &other);  
  
 friend bool operator==(const Hexagon &a, const Hexagon &b);  
  
 friend std::istream &operator>>(std::istream &is, Hexagon &hexagon);  
  
 friend std::ostream &operator<<(std::ostream &os, const Hexagon &hexagon);  
  
 void \*operator new(size\_t sz);  
  
 void operator delete(void \*ptr, size\_t sz) noexcept;  
  
 ~Hexagon();  
};  
  
  
#endif //LAB6\_HEXAGON\_H

hexagon.cpp

#include "hexagon.h"  
  
  
Hexagon::Hexagon() {}  
  
TAllocator Hexagon::alloc(sizeof(Hexagon), 10);  
  
double Hexagon::Area() {  
 const double eps = 1e-9;  
 bool convex = true;  
 for (size\_t i = 0; i < size - 1; ++i) {  
 Point a = P[(i + 1) % size] - P[i];  
 Point b = P[(i + 2) % size] - P[(i + 1) % size];  
 if (a \* b > eps) {  
 convex = false;  
 }  
 }  
 if (!convex) {  
 return -1.0;  
 }  
 double area = 0.0;  
 for (size\_t i = 0; i < size; ++i) {  
 area += dist(P[i], P[(i + 1) % size]);  
 }  
 if (area < 0.0)area = -1.0 \* area;  
 return area;  
}  
  
size\_t Hexagon::VertexesNumber() {  
 return size;  
}  
  
Hexagon &Hexagon::operator=(const Hexagon &other) {  
 for (size\_t i = 0; i < size; ++i) {  
 this->P[i] = other.P[i];  
 }  
 return \*this;  
}  
  
bool operator==(const Hexagon &a, const Hexagon &b) {  
 for (size\_t i = 0; i < Hexagon::size; ++i) {  
 if (!(a.P[i] == b.P[i]))return false;  
 }  
 return true;  
}  
  
std::ostream &operator<<(std::ostream &os, const Hexagon &hexagon) {  
 os << "Hexagon: ";  
 for (size\_t i = 0; i < Hexagon::size; ++i) {  
 os << hexagon.P[i];  
 }  
 os << "\n";  
 return os;  
}  
  
std::istream &operator>>(std::istream &is, Hexagon &hexagon) {  
 for (size\_t i = 0; i < hexagon.size; ++i) {  
 is >> hexagon.P[i];  
 }  
 return is;  
}  
  
void \*Hexagon::operator new(size\_t sz) {  
 return alloc.allocate(sz);  
}  
  
void Hexagon::operator delete(void \*ptr, size\_t sz) noexcept {  
 alloc.deallocate(ptr);  
}  
  
Hexagon::~Hexagon() {}

point.h

#ifndef LAB6\_POINT\_H  
#define LAB6\_POINT\_H  
  
#include <iostream>  
  
class Point {  
private:  
 double \_x, \_y;  
public:  
 Point();  
  
 Point(double x, double y);  
  
 friend bool operator==(const Point &a, const Point &b);  
  
 friend double operator\*(const Point &a, const Point &b);  
  
 friend const Point operator-(const Point &a, const Point &b);  
  
 friend double dist(const Point &a, const Point &b);  
  
 friend std::istream &operator>>(std::istream &is, Point &p);  
  
 friend std::ostream &operator<<(std::ostream &os, const Point &p);  
};  
  
  
#endif //LAB6\_POINT\_H

point.cpp

#include "point.h"  
  
Point::Point() : \_x(0.0), \_y(0.0) {}  
  
Point::Point(double x, double y) : \_x(x), \_y(y) {}  
  
double operator\*(const Point &a, const Point &b) {  
 return a.\_x \* b.\_y - b.\_x \* a.\_y;  
}  
  
const Point operator-(const Point &a, const Point &b) {  
 Point c;  
 c.\_x = b.\_x - a.\_x;  
 c.\_y = b.\_y - a.\_y;  
 return c;  
}  
  
double dist(const Point &a, const Point &b) {  
 double dx = (b.\_x - a.\_x);  
 double mid = (b.\_y + a.\_y) / 2.0;  
 return dx \* mid;  
}  
  
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;  
}  
  
bool operator==(const Point &a, const Point &b) {  
 return (a.\_x == b.\_x && a.\_y == b.\_y);  
}

tlinkedlist\_item.h

#ifndef LAB6\_TLINKEDLIST\_ITEM\_H  
#define LAB6\_TLINKEDLIST\_ITEM\_H  
  
#include "hexagon.h"  
#include "iostream"  
#include "memory"  
  
using std::shared\_ptr;  
using std::make\_shared;  
  
template <typename T>  
class TLinkedListItem {  
private:  
 shared\_ptr<T> val;  
 shared\_ptr<TLinkedListItem<T>> next;  
public:  
 TLinkedListItem(shared\_ptr<T> hexagon, shared\_ptr<TLinkedListItem<T>> nxt);  
  
 void SetNext(shared\_ptr<TLinkedListItem<T>> nxt);  
  
 shared\_ptr<TLinkedListItem<T>> GetNext();  
  
 shared\_ptr<T> GetVal();  
  
 template<typename X>  
 friend std::ostream &operator<<(std::ostream &os, const TLinkedListItem<X> &item);  
  
 virtual ~TLinkedListItem();  
};  
  
  
#endif //LAB6\_TLINKEDLIST\_ITEM\_H

tlinkedlist\_item.cpp

#include "tlinkedlist\_item.h"  
  
template<typename T>  
TLinkedListItem<T>::~TLinkedListItem() {  
 printf("DELETED ITEM\n");  
}  
  
template<typename T>  
TLinkedListItem<T>::TLinkedListItem(shared\_ptr<T> figure, shared\_ptr<TLinkedListItem<T>> nxt) {  
 val = figure;  
 next = nxt;  
}  
  
template<typename T>  
shared\_ptr<TLinkedListItem<T>> TLinkedListItem<T>::GetNext() {  
 return next;  
}  
  
template<typename T>  
void TLinkedListItem<T>::SetNext(shared\_ptr<TLinkedListItem<T>> nxt) {  
 next = nxt;  
}  
  
template<typename T>  
shared\_ptr<T> TLinkedListItem<T>::GetVal() {  
 return val;  
}  
  
template<typename T>  
std::ostream &operator<<(std::ostream &os, const TLinkedListItem<T> &item) {  
 os << \*item.val;  
 return os;  
}  
  
  
template class TLinkedListItem<Hexagon>;  
template std::ostream &operator<<(std::ostream &os, const TLinkedListItem<Hexagon> &item) ;

tlinkedlist.h

#ifndef LAB6\_TLINKEDLIST\_H  
#define LAB6\_TLINKEDLIST\_H  
  
#include "tlinkedlist\_item.h"  
#include "titerator.h"  
  
template<typename T>  
class TLinkedList {  
private:  
 size\_t len;  
 shared\_ptr<TLinkedListItem<T>> head;  
public:  
 TLinkedList();  
  
 TLinkedList(const TLinkedList<T> &list);  
  
 shared\_ptr<T> First();  
  
 shared\_ptr<T> Last();  
  
 TIterator<TLinkedListItem<T>, T> begin();  
  
 TIterator<TLinkedListItem<T>, T> end();  
  
 void InsertFirst(shared\_ptr<T> hexagon);  
  
 void InsertLast(shared\_ptr<T> hexagon);  
  
 void Insert(shared\_ptr<T> hexagon, size\_t pos);  
  
 void RemoveFirst();  
  
 void RemoveLast();  
  
 void Remove(size\_t pos);  
  
 shared\_ptr<T> GetItem(size\_t ind);  
  
 bool Empty();  
  
 size\_t Length();  
  
 template<typename X>  
 friend std::ostream &operator<<(std::ostream &os, const TLinkedList<X> &list);  
  
 void Clear();  
  
 virtual ~TLinkedList();  
};  
  
  
#endif //LAB6\_TLINKEDLIST\_H

tlinkedlist.cpp

#include "tlinkedlist.h"  
  
template<typename T>  
TLinkedList<T>::TLinkedList() {  
 len = 0;  
 head = nullptr;  
}  
  
template<typename T>  
TLinkedList<T>::TLinkedList(const TLinkedList<T> &list) {  
 len = list.len;  
 if (!list.len) {  
 head = nullptr;  
 return;  
 }  
 head = make\_shared<TLinkedListItem<T>>(list.head->GetVal(), nullptr);  
 shared\_ptr<TLinkedListItem<T>> cur = head;  
 shared\_ptr<TLinkedListItem<T>> it = list.head;  
 for (size\_t i = 0; i < len - 1; ++i) {  
 it = it->GetNext();  
 shared\_ptr<TLinkedListItem<T>> new\_item = make\_shared<TLinkedListItem<T>>(it->GetVal(), nullptr);  
 cur->SetNext(new\_item);  
 cur = cur->GetNext();  
 }  
}  
  
template<typename T>  
shared\_ptr<T> TLinkedList<T>::First() {  
 if (len == 0) {  
 return nullptr;  
 }  
 return head->GetVal();  
}  
  
template<typename T>  
shared\_ptr<T> TLinkedList<T>::Last() {  
 if (len == 0) {  
 return nullptr;  
 }  
 shared\_ptr<TLinkedListItem<T>> cur = head;  
 for (size\_t i = 0; i < len - 1; ++i) {  
 cur = cur->GetNext();  
 }  
 return cur->GetVal();  
}  
  
template<typename T>  
void TLinkedList<T>::InsertFirst(shared\_ptr<T> figure) {  
 shared\_ptr<TLinkedListItem<T>> it = make\_shared<TLinkedListItem<T>>(figure, head);  
 head = it;  
 len++;  
}  
  
template<typename T>  
void TLinkedList<T>::InsertLast(shared\_ptr<T> figure) {  
 if (len == 0) {  
 head = make\_shared<TLinkedListItem<T>>(figure, nullptr);  
 len = 1;  
 return;  
 }  
 shared\_ptr<TLinkedListItem<T>> cur = head;  
 for (size\_t i = 0; i < len - 1; ++i) {  
 cur = cur->GetNext();  
 }  
 shared\_ptr<TLinkedListItem<T>> it = make\_shared<TLinkedListItem<T>>(figure, nullptr);  
 cur->SetNext(it);  
 len++;  
}  
  
template<typename T>  
void TLinkedList<T>::Insert(shared\_ptr<T> figure, size\_t pos) {  
 if (pos > len || pos < 0) {  
 return;  
 }  
 shared\_ptr<TLinkedListItem<T>> cur = head;  
 shared\_ptr<TLinkedListItem<T>> prev = nullptr;  
 for (size\_t i = 0; i < pos; ++i) {  
 prev = cur;  
 cur = cur->GetNext();  
 }  
 shared\_ptr<TLinkedListItem<T>> it = make\_shared<TLinkedListItem<T>>(figure, cur);  
 if (prev) {  
 prev->SetNext(it);  
 } else {  
 head = it;  
 }  
 len++;  
}  
  
template<typename T>  
void TLinkedList<T>::RemoveFirst() {  
 if (!len)return;  
 shared\_ptr<TLinkedListItem<T>> del = head;  
 head = head->GetNext();  
 len--;  
}  
  
template<typename T>  
void TLinkedList<T>::RemoveLast() {  
 if (!len)return;  
 if (len == 1) {  
 head = nullptr;  
 len = 0;  
 return;  
 }  
 shared\_ptr<TLinkedListItem<T>> cur = head;  
 for (size\_t i = 0; i < len - 2; ++i) {  
 cur = cur->GetNext();  
 }  
 shared\_ptr<TLinkedListItem<T>> del = cur->GetNext();  
 cur->SetNext(nullptr);  
 len--;  
}  
  
template<typename T>  
void TLinkedList<T>::Remove(size\_t pos) {  
 if (!len)return;  
 if (pos < 0 || pos >= len)return;  
 shared\_ptr<TLinkedListItem<T>> cur = head;  
 shared\_ptr<TLinkedListItem<T>> prev = nullptr;  
 for (size\_t i = 0; i < pos; ++i) {  
 prev = cur;  
 cur = cur->GetNext();  
 }  
 if (prev) {  
 prev->SetNext(cur->GetNext());  
 } else {  
 head = cur->GetNext();  
 }  
 len--;  
}  
  
template<typename T>  
shared\_ptr<T> TLinkedList<T>::GetItem(size\_t ind) {  
 if (ind < 0 || ind >= len)return nullptr;  
 shared\_ptr<TLinkedListItem<T>> cur = head;  
 for (size\_t i = 0; i < ind; ++i) {  
 cur = cur->GetNext();  
 }  
 return cur->GetVal();  
}  
  
template<typename T>  
bool TLinkedList<T>::Empty() {  
 return len == 0;  
}  
  
template<typename T>  
size\_t TLinkedList<T>::Length() {  
 return len;  
}  
  
template<typename T>  
std::ostream &operator<<(std::ostream &os, const TLinkedList<T> &list) {  
 shared\_ptr<TLinkedListItem<T>> cur = list.head;  
 os << "List: \n";  
 for (size\_t i = 0; i < list.len; ++i) {  
 os << \*cur;  
 cur = cur->GetNext();  
 }  
 return os;  
}  
  
template<typename T>  
void TLinkedList<T>::Clear() {  
 while (!(this->Empty())) {  
 this->RemoveFirst();  
 }  
}  
  
template<typename T>  
TLinkedList<T>::~TLinkedList() {  
 while (!(this->Empty())) {  
 this->RemoveFirst();  
 }  
 printf("DELETED LIST\n");  
}  
  
template<typename T>  
TIterator<TLinkedListItem<T>, T> TLinkedList<T>::begin() {  
 return TIterator<TLinkedListItem<T>, T>(head);  
}  
  
template<typename T>  
TIterator<TLinkedListItem<T>, T> TLinkedList<T>::end() {  
 TIterator<TLinkedListItem<T>, T> it = begin();  
 for (size\_t i = 0; i < len; ++i) {  
 it++;  
 }  
 return it;  
}  
  
template  
class TLinkedList<Hexagon>;  
  
template std::ostream &operator<<(std::ostream &os, const TLinkedList<Hexagon> &list);

titerator.h

#ifndef LAB6\_TITERATOR\_H  
#define LAB6\_TITERATOR\_H  
  
#include "iostream"  
#include "memory"  
  
using std::shared\_ptr;  
  
template<typename node, typename T>  
class TIterator {  
public:  
 TIterator(shared\_ptr<node> other) {  
 ptr = other;  
 }  
  
 shared\_ptr<T> operator\*() {  
 return ptr->GetVal();  
 }  
  
 shared\_ptr<T> operator->() {  
 return ptr->GetVal();  
 }  
  
 TIterator<node, T> operator++() {  
 return ptr = ptr->GetNext();  
 }  
  
 TIterator<node, T> operator++(int) {  
 TIterator iter(\*this);  
 ++(\*this);  
 return iter;  
 }  
  
 bool operator==(TIterator<node, T> const &other) {  
 return ptr == other.ptr;  
 }  
  
 bool operator!=(TIterator<node, T> const &other) {  
 return !(\*this == other);  
 }  
  
private:  
 shared\_ptr<node> ptr;  
};  
  
#endif //LAB6\_TITERATOR\_H

tvector\_item.h

#ifndef LAB6\_TVECTOR\_ITEM\_H  
#define LAB6\_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 //LAB6\_TVECTOR\_ITEM\_H

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 \*>;

tvector.h

#ifndef LAB6\_TVECTOR\_H  
#define LAB6\_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 //LAB6\_TVECTOR\_H

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 \*>;

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

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(block\_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);  
}