МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ  
ФЕДЕРАЦИИ МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

ЛАБОРАТОРНАЯ РАБОТА №8 по курсу объектно-ориентированное программирование I семестр, 2021/22 уч. год

Студент *Борисов Ян Артурович, группа М80-208Б-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 (контейнер 2-го уровня, согласно варианту задания). 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.

**Дневник отладки**

Во время выполнения лабораторной были некие трудности с реализацией аллокатора, позже они были полностью ликвидированы.

**Недочёты**  
Недочётов не было обнаружено.

**Выводы**

Лабораторная работа №8 позволила мне реализовать свой класс аллокаторов, полностью прочувствовать процесс выделения памяти на низкоуровневых языках программирования. Лабораторная прошла успешно.

**Исходный код**

TAllocatorBlock.h  
  
#ifndef TALLOCATORBLOCK\_H

#define TALLOCATORBLOCK\_H

#include "TLinkedList.h"

#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.Insert(malloc(size));

}

}

void\* Allocate(const size\_t& size){

if(size != this->size){

std::cout << "Error during allocation\n";

}

if(unused\_blocks.Length()){

for(int i = 0; i < 5; ++i){

unused\_blocks.Insert(malloc(size));

}

}

void\* tmp = unused\_blocks.GetItem(1);

used\_blocks.Insert(unused\_blocks.GetItem(1));

unused\_blocks.Remove(0);

return tmp;

}

void Deallocate(void\* ptr){

unused\_blocks.Insert(ptr);

}

~TAllocatorBlock(){

while(used\_blocks.size()){

try{

free(used\_blocks.GetItem(1);)

used\_blocks.Remove(0);

} catch(...){

used\_blocks.Remove(0);

}

}

while(unused\_blocks.size()){

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

HListItem.cpp  
  
#include <iostream>  
#include "HListItem.h"  
  
template <class T> HListItem<T>::HListItem(const std::shared\_ptr<Square> &square) {  
 this->square = square;  
 this->next = nullptr;  
}  
template <class A> std::ostream& operator<<(std::ostream& os,HListItem<A> &obj) {  
 os << "[" << obj.square << "]" << std::endl;  
 return os;  
}  
template <class T> HListItem<T>::~HListItem() {  
}

HListItem.h  
  
#ifndef HLISTITEM\_H  
#define HLISTITEM\_H  
#include <iostream>  
#include "square.h"  
#include <memory>  
  
template <class T> class HListItem {  
public:  
 HListItem(const std::shared\_ptr<Square> &square);  
 template <class A> friend std::ostream& operator<<(std::ostream& os, HListItem<A> &obj);  
 ~HListItem();  
 std::shared\_ptr<T> square;  
 std::shared\_ptr<HListItem<T>> next;  
};  
#include "HListItem.cpp"  
#endif

TLinkedList.cpp

#include <iostream>  
#include "TLinkedList.h"  
   
template <class T> TLinkedList<T>::TLinkedList() {  
 size\_of\_list = 0;  
 std::shared\_ptr<HListItem<T>> front;  
 std::shared\_ptr<HListItem<T>> back;  
 std::cout << "Square List created" << std::endl;  
}  
template <class T> TLinkedList<T>::TLinkedList(const std::shared\_ptr<TLinkedList> &other){  
 front = other->front;  
 back = other->back;  
}  
template <class T> size\_t TLinkedList<T>::Length() {  
 return size\_of\_list;  
}  
template <class T> bool TLinkedList<T>::Empty() {  
 return size\_of\_list;  
}  
template <class T> std::shared\_ptr<Square>& TLinkedList<T>::GetItem(size\_t idx){  
 int k = 0;  
 std::shared\_ptr<HListItem<T>> obj = front;  
 while (k != idx){  
 k++;  
 obj = obj->next;  
 }  
 return obj->square;  
}  
template <class T> std::shared\_ptr<Square>& TLinkedList<T>::First() {  
 return front->square;  
}  
template <class T> std::shared\_ptr<Square>& TLinkedList<T>::Last() {  
 return back->square;  
}  
template <class T> void TLinkedList<T>::InsertLast(const std::shared\_ptr<Square> &&square) {  
 std::shared\_ptr<HListItem<T>> obj (new HListItem<T>(square));  
 if(size\_of\_list == 0) {  
 front = obj;  
 back = obj;  
 size\_of\_list++;  
 return;  
 }  
 back->next = obj;  
 back = obj;  
 obj->next = nullptr;  
 size\_of\_list++;  
}  
template <class T> void TLinkedList<T>::RemoveLast() {  
 if (size\_of\_list == 0) {  
 std::cout << "Square does not pop\_back, because the Square List is empty" << std:: endl;  
 } else {  
 if (front == back) {  
 RemoveFirst();  
 size\_of\_list--;  
 return;  
 }  
 std::shared\_ptr<HListItem<T>> prev\_del = front;  
 while (prev\_del->next != back) {  
 prev\_del = prev\_del->next;  
 }  
 prev\_del->next = nullptr;  
 back = prev\_del;  
 size\_of\_list--;  
 }   
}  
template <class T> void TLinkedList<T>::InsertFirst(const std::shared\_ptr<Square> &&square) {  
 std::shared\_ptr<HListItem<T>> obj (new HListItem<T>(square));  
 if(size\_of\_list == 0) {  
 front = obj;  
 back = obj;  
 } else {  
 obj->next = front;  
 front = obj;  
 }  
 size\_of\_list++;  
}  
template <class T> void TLinkedList<T>::RemoveFirst() {  
 if (size\_of\_list == 0) {  
 std::cout << "Square does not pop\_front, because the Square List is empty" << std:: endl;  
 } else {  
 std::shared\_ptr<HListItem<T>> del = front;  
 front = del->next;  
 size\_of\_list--;  
 }  
}  
template <class T> void TLinkedList<T>::Insert(const std::shared\_ptr<Square> &&square, size\_t position) {  
 if (position <0) {  
 std::cout << "Position < zero" << std::endl;  
 } else if (position > size\_of\_list) {  
 std::cout << " Position > size\_of\_list" << std::endl;  
 } else {  
 std::shared\_ptr<HListItem<T>> obj (new HListItem<T>(square));  
 if (position == 0) {  
 front = obj;  
 back = obj;  
 } else {  
 int k = 0;  
 std::shared\_ptr<HListItem<T>> prev\_insert = front;  
 std::shared\_ptr<HListItem<T>> next\_insert;  
 while(k+1 != position) {  
 k++;  
 prev\_insert = prev\_insert->next;  
 }  
 next\_insert = prev\_insert->next;  
 prev\_insert->next = obj;  
 obj->next = next\_insert;  
 }  
 size\_of\_list++;  
 }  
}  
template <class T> void TLinkedList<T>::Remove(size\_t position) {  
 if (position > size\_of\_list ) {  
 std:: cout << "Position " << position << " > " << "size " << size\_of\_list << " Not correct erase" << std::endl;  
 } else if (position < 0) {  
 std::cout << "Position < 0" << std::endl;  
 } else {  
 if (position == 0) {  
 RemoveFirst();  
 } else {  
 int k = 0;  
 std::shared\_ptr<HListItem<T>> prev\_erase = front;  
 std::shared\_ptr<HListItem<T>> next\_erase;  
 std::shared\_ptr<HListItem<T>> del;F  
 while( k+1 != position) {  
 k++;  
 prev\_erase = prev\_erase->next;  
 }  
 next\_erase = prev\_erase->next;  
 del = prev\_erase->next;  
 next\_erase = del->next;  
 prev\_erase->next = next\_erase;  
 }  
 size\_of\_list--;  
 }  
}  
template <class T> void TLinkedList<T>::Clear() {  
 std::shared\_ptr<HListItem<T>> del = front;  
 std::shared\_ptr<HListItem<T>> prev\_del;  
 if(size\_of\_list !=0 ) {  
 while(del->next != nullptr) {  
 prev\_del = del;  
 del = del->next;  
 }  
 size\_of\_list = 0;  
 // std::cout << "HListItem deleted" << std::endl;  
 }   
 size\_of\_list = 0;  
 std::shared\_ptr<HListItem<T>> front;  
 std::shared\_ptr<HListItem<T>> back;  
}  
template <class T> std::ostream& operator<<(std::ostream& os, TLinkedList<T>& hl) {  
 if (hl.size\_of\_list == 0) {  
 os << "The square list is empty, so there is nothing to output" << std::endl;  
 } else {  
 os << "Print Square List" << std::endl;  
 std::shared\_ptr<HListItem<T>> obj = hl.front;  
 while(obj != nullptr) {  
 if (obj->next != nullptr) {  
 os << obj->square << " " << "," << " ";  
 obj = obj->next;  
 } else {  
 os << obj->square;  
 obj = obj->next;  
 }  
 }  
 os << std::endl;  
 }  
 return os;  
}  
template <class T> TLinkedList<T>::~TLinkedList() {  
 std::shared\_ptr<HListItem<T>> del = front;  
 std::shared\_ptr<HListItem<T>> prev\_del;  
 if(size\_of\_list !=0 ) {  
 while(del->next != nullptr) {  
 prev\_del = del;  
 del = del->next;  
 }  
 size\_of\_list = 0;  
 std::cout << "Square List deleted" << std::endl;  
 }   
}

TLinkedList.h  
  
#ifndef HLIST\_H  
#define HLIST\_H  
#include <iostream>  
#include "HListItem.h"  
#include "square.h"  
#include <memory>  
  
template <class T> class TLinkedList {  
public:  
 TLinkedList();  
 int size\_of\_list;  
 size\_t Length();  
 std::shared\_ptr<Square>& First();  
 std::shared\_ptr<Square>& Last();  
 std::shared\_ptr<Square>& GetItem(size\_t idx);  
 bool Empty();  
 TLinkedList(const std::shared\_ptr<TLinkedList> &other);  
 void InsertFirst(const std::shared\_ptr<Square> &&square);  
 void InsertLast(const std::shared\_ptr<Square> &&square);  
 void RemoveLast();  
 void RemoveFirst();  
 void Insert(const std::shared\_ptr<Square> &&square, size\_t position);  
 void Remove(size\_t position);  
 void Clear();  
 template <class A> friend std::ostream& operator<<(std::ostream& os, TLinkedList<A>& list);  
 ~TLinkedList();   
private:  
 std::shared\_ptr<HListItem<T>> front;  
 std::shared\_ptr<HListItem<T>> back;  
};  
#include "TLinkedList.cpp"  
#endif