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<Pentagon> &pentagon) {
  this->pentagon = pentagon;
  this->next = nullptr;
}
template <class A> std::ostream& operator<<(std::ostream& os,HListItem<A> &obj) {
  os << "[" << obj.pentagon << "]" << std::endl;
  return os;
}
template <class T> HListItem<T>::~HListItem() {
}
```

HListItem.h

```
#ifndef HLISTITEM_H
#define HLISTITEM_H
#include <iostream>
#include "pentagon.h"
#include <memory>

template <class T> class HListItem {
public:
```

```
  HListItem(const std::shared_ptr<Pentagon> &pentagon);
  template <class A> friend std::ostream& operator<<(std::ostream& os, HListItem<A>
&obj);
  ~HListItem();
  std::shared_ptr<T> pentagon;
  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 << "Pentagon 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<Pentagon>& 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->pentagon;
}
template <class T> std::shared_ptr<Pentagon>& TLinkedList<T>::First() {
  return front->pentagon;
}
template <class T> std::shared_ptr<Pentagon>& TLinkedList<T>::Last() {
```

```
  return back->pentagon;
}
template <class T> void TLinkedList<T>::InsertLast(const std::shared_ptr<Pentagon>
&&pentagon) {
  std::shared_ptr<HListItem<T>> obj (new HListItem<T>(pentagon));
  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 << "Pentagon does not pop_back, because the Pentagon 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<Pentagon>
&&pentagon) {
    std::shared_ptr<HListItem<T>> obj (new HListItem<T>(pentagon));
    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 << "Pentagon does not pop_front, because the Pentagon 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<Pentagon>
&&pentagon, 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>(pentagon));
    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 pentagon list is empty, so there is nothing to output" << std::endl;
  } else {
    os << "Print Pentagon List" << std::endl;
    std::shared_ptr<HListItem<T>> obj = hl.front;
    while(obj != nullptr) {
      if (obj->next != nullptr) {
        os << obj->pentagon << " " << "," << " ";
        obj = obj->next;
      } else {
```

```
        os << obj->pentagon;
        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 << "Pentagon List deleted" << std::endl;
  }
}
```

TLinkedList.h

```
#ifndef HLIST_H
#define HLIST_H
#include <iostream>
#include "HListItem.h"
#include "pentagon.h"
#include <memory>

template <class T> class TLinkedList {
public:
  TLinkedList();
  int size_of_list;
  size_t Length();
  std::shared_ptr<Pentagon>& First();
  std::shared_ptr<Pentagon>& Last();
  std::shared_ptr<Pentagon>& GetItem(size_t idx);
  bool Empty();
  TLinkedList(const std::shared_ptr<TLinkedList> &other);
  void InsertFirst(const std::shared_ptr<Pentagon> &&pentagon);
  void InsertLast(const std::shared_ptr<Pentagon> &&pentagon);
  void RemoveLast();
  void RemoveFirst();
  void Insert(const std::shared_ptr<Pentagon> &&pentagon, 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
```