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 *Каширин Кирилл Дмитриевич, группа М8О-208Б-20*

Преподаватель *Дорохов Евгений Павлович*

Условие

- Вариант 7:
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. Алокатор 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.

Исходный код лежит в 11 файлах:

1. main.cpp: основная программа, взаимодействие с пользователем посредством команд из меню
2. figure.h: описание абстрактного класса фигур
3. point.h: описание класса точки
4. hlist_item.h: описание класса элемента списка восьмиугольника
5. tlinkedlist.h: описание класса связанного списка
6. hexagon.h: описание класса восьмиугольника, наследующегося от figures
7. point.cpp: реализация класса точки
8. hexagon.cpp: реализация класса восьмиугольника, наследующегося от figures
9. tlinkedlist.cpp: реализация класса связанного списка
10. hlist_item.cpp: реализация класса элемента связанного списка
11. Iterator.h: реализация класса итератора связанного списка
12. tallocation_block.h: реализация класса алокатора связанного списка
13. TVector.h: реализация класса шаблонного вектора для использования в аллокаторе
14. TVector_item.h: реализация класса элемента шаблонного вектора для использования в аллокаторе

Дневник отладки

Проблем и ошибок при написании данной работы не возникло.

Недочёты

Выводы

В данной лабораторной работе я на практике познакомился с понятием аллокатора. Поскольку аллокаторы используются почти во всех структурах данных, то это очень важная тема, которую должен знать каждый программист на C++.

Исходный код:

figure.h

```
#ifndef FIGURE_H
#define FIGURE_H
#include <iostream>
#include "point.h"
class Figure {
public:
    virtual double Area() = 0;
    virtual size_t VertexesNumber() = 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);

  friend std::istream& operator>>(std::istream& is, Point& p);
  friend std::ostream& operator<<(std::ostream& os, Point& p);

  double x();
  double y();

private:
  double x_;
  double y_;
};

#endif // POINT_H
```

point.cpp

```cpp
#include "point.h"
#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_;
}

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;
}

double Point::x(){
  return x_;
}

double Point::y(){
  return y_;
}
```

hexagon.h

```cpp
#ifndef HEXAGON_H
#define HEXAGON_H
#include <iostream>
#include "figure.h"
```

```cpp
#include "point.h"
#include <memory>

class Hexagon : public Figure {
public:
    Hexagon();
    Hexagon(std::istream &is);
    Hexagon(Point a, Point b, Point c, Point d, Point e, Point f);
    Hexagon(std::shared_ptr<Hexagon>& other);
    double Area();
    size_t VertexesNumber();
    virtual ~Hexagon();
    Hexagon& operator=(const Hexagon& other);
    Hexagon& operator==(const Hexagon& other);
    friend std::ostream& operator<<(std::ostream& os, std::shared_ptr<Hexagon>& h);
private:
    Point a, b, c, d, e, f;
};

#endif // HEXAGON_H
```

hexagon.cpp

```cpp
#include <iostream>
#include "hexagon.h"
#include <cmath>

Hexagon::Hexagon(): a(0,0),b(0,0),c(0,0),d(0,0),e(0,0),f(0,0) {
}
Hexagon::Hexagon(std::istream &is) {
    is >> a;
    is >> b;
    is >> c;
    is >> d;
    is >> e;
    is >> f;
}
Hexagon::Hexagon(Point a1, Point b1,Point c1, Point d1, Point e1, Point f1): a(a1),b(b1)
}
double Hexagon::Area() {
    return 0.5*abs(a.x()*b.y()+b.x()*c.y()+c.x()*d.y()+d.x()*e.y()+e.x()*f.y()+f.x()*a.y
}
Hexagon::~Hexagon() {
```

```
}
size_t Hexagon::VertexesNumber() {
    return 6;
}
Hexagon::Hexagon(std::shared_ptr<Hexagon>& other):Hexagon(other->a,other->b,other->c,oth
}
Hexagon& Hexagon::operator = (const Hexagon& other) {
  if (this == &other) return *this;
  a = other.a;
  b = other.b;
  c = other.c;
  d = other.d;
  e = other.e;
  f = other.f;
  //std::cout << "Hexagon copied" << std::endl;
  return *this;
}
Hexagon& Hexagon::operator == (const Hexagon& other) {
  if (this == &other){
    std::cout << "Hexagons are equal" << std::endl;
  } else {
    std::cout << "Hexagons are not equal" << std::endl;
  }
}
std::ostream& operator<<(std::ostream& os, std::shared_ptr<Hexagon>& h) {
  os << h->a << h->b << h->c << h->d << h->e << h->f;
  return os;
}
```

hlist_item.h

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

template <class T> class HListItem {
public:
  HListItem(const std::shared_ptr<Hexagon> &hexagon);
  template <class A> friend std::ostream& operator<<(std::ostream& os, HListItem<A> &obj
  ~HListItem();
  std::shared_ptr<T> hexagon;
```

```
  std::shared_ptr<HListItem<T>> next;
  std::shared_ptr<HListItem<T>> SetNext(std::shared_ptr<HListItem<T>> &next_);
  std::shared_ptr<HListItem<T>> GetNext();
  std::shared_ptr<T>& GetValue();
};
#include "hlist_item.inl"
#endif //HLISTITEM_H
```

hlist_item.inl

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

template <class T> HListItem<T>::HListItem(const std::shared_ptr<Hexagon> &hexagon) {
  this->hexagon = hexagon;
  this->next = nullptr;
}
template <class T> std::shared_ptr<HListItem<T>> HListItem<T>::SetNext(std::shared_ptr<H
  std::shared_ptr<HListItem<T>> prev = this->next;
  this->next = next_;
  return prev;
}
template <class T> std::shared_ptr<T>& HListItem<T>::GetValue() {
  return this->hexagon;
}
template <class T> std::shared_ptr<HListItem<T>> HListItem<T>::GetNext() {
  return this->next;
}
template <class A> std::ostream& operator<<(std::ostream& os,HListItem<A> &obj) {
  os << "[" << obj.hexagon << "]" << std::endl;
  return os;
}
template <class T> HListItem<T>::~HListItem() {
}
```

tlinkedlist.h

```
#ifndef HLIST_H
#define HLIST_H
#include <iostream>
#include "hlist_item.h"
#include "hexagon.h"
#include <memory>
```

```
#include "titerator.h"

template <class T> class TLinkedList {
public:
  TLinkedList();
  int size_of_list;
  size_t Length();
  std::shared_ptr<T>& First();
  std::shared_ptr<Hexagon>& Last();
  std::shared_ptr<Hexagon>& GetItem(size_t idx);
  bool Empty();
  TLinkedList(const std::shared_ptr<TLinkedList> &other);
  void InsertFirst(const std::shared_ptr<Hexagon> &&hexagon);
  void InsertLast(const std::shared_ptr<Hexagon> &&hexagon);
  void RemoveLast();
  void RemoveFirst();
  void Insert(const std::shared_ptr<Hexagon> &&hexagon, size_t position);
  void Remove(size_t position);
  void Clear();
  template <class A> friend std::ostream& operator<<(std::ostream& os, TLinkedList<A>& l
  ~TLinkedList();
  Titerator<HListItem<T>, T> begin();
  Titerator<HListItem<T>, T> end();
private:
  std::shared_ptr<HListItem<T>> front;
  std::shared_ptr<HListItem<T>> back;
};
#include "tlinkedlist.inl"
#endif //HList_H
```

tlinkedlist.inl

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

template <class T>
Titerator<HListItem<T>, T> TLinkedList<T>::begin() {
  return Titerator<HListItem<T>, T> (front);
}

template <class T>
Titerator<HListItem<T>, T> TLinkedList<T>::end() {
  return Titerator<HListItem<T>, T>(back);
```

```
}

template <class T> TLinkedList<T>::TLinkedList() {
  size_of_list = 0;
  std::shared_ptr<HListItem<T>> front = nullptr;
  std::shared_ptr<HListItem<T>> back = nullptr;
  std::cout << "Hexagon 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<Hexagon>& TLinkedList<T>::GetItem(size_t idx){
  int k = 0;
  std::shared_ptr<HListItem<T>> obj = front;
  while (k != idx){
    k++;
    obj = obj->GetNext();
  }
  return obj->GetValue();
}
template <class T> std::shared_ptr<T>& TLinkedList<T>::First() {
    return front->GetValue();
}
template <class T> std::shared_ptr<Hexagon>& TLinkedList<T>::Last() {
  return back->GetValue();
}
template <class T> void TLinkedList<T>::InsertLast(const std::shared_ptr<Hexagon> &&hexa
  std::shared_ptr<HListItem<T>> obj (new HListItem<T>(hexagon));
 // std::shared_ptr<HListItem<T>> obj = std::make_shared<HListItem<T>>(HListItem<T>(hexa
  if(size_of_list == 0) {
    front = obj;
    back = obj;
    size_of_list++;
    return;
  }
```

```
  back->SetNext(obj); // = obj;
  back = obj;
  obj->next = nullptr; // = nullptr;
  size_of_list++;
}
template <class T> void TLinkedList<T>::RemoveLast() {
  if (size_of_list == 0) {
    std::cout << "Hexagon does not pop_back, because the Hexagon List is empty" << std::
  } else {
    if (front == back) {
      RemoveFirst();
      size_of_list--;
      return;
    }
    std::shared_ptr<HListItem<T>> prev_del = front;
    while (prev_del->GetNext() != back) {
      prev_del = prev_del->GetNext();
    }
    prev_del->next = nullptr;
    back = prev_del;
    size_of_list--;
    }
}
template <class T> void TLinkedList<T>::InsertFirst(const std::shared_ptr<Hexagon> &&hex
    std::shared_ptr<HListItem<T>> obj (new HListItem<T>(hexagon));
    if(size_of_list == 0) {
      front = obj;
      back = obj;
    } else {
      obj->SetNext(front); // = front;
      front = obj;
    }
    size_of_list++;
}
template <class T> void TLinkedList<T>::RemoveFirst() {
    if (size_of_list == 0) {
      std::cout << "Hexagon does not pop_front, because the Hexagon List is empty" << st
    } else {
    std::shared_ptr<HListItem<T>> del = front;
    front = del->GetNext();
    size_of_list--;
    }
```

```
}
template <class T> void TLinkedList<T>::Insert(const std::shared_ptr<Hexagon> &&hexagon,
  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>(hexagon));
    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->GetNext();
      }
      next_insert = prev_insert->GetNext();
      prev_insert->SetNext(obj); // = obj;
      obj->SetNext(next_insert); // = 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 c
  } 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;
      while( k+1 != position) {
        k++;
        prev_erase = prev_erase->GetNext();
```

```
      }
      next_erase = prev_erase->GetNext();
      del = prev_erase->GetNext();
      next_erase = del->GetNext();
      prev_erase->SetNext(next_erase); // = 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->GetNext() != nullptr) {
      prev_del = del;
      del = del->GetNext();
    }
    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 hexagon list is empty, so there is nothing to output" << std::endl;
  } else {
    os << "Print Hexagon List" << std::endl;
    std::shared_ptr<HListItem<T>> obj = hl.front;
    while(obj != nullptr) {
      if (obj->GetNext() != nullptr) {
        os << obj->GetValue() << " " << "," << " ";
        obj = obj->GetNext();
      } else {
        os << obj->GetValue();
        obj = obj->GetNext();
      }
    }
    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->GetNext() != nullptr) {
      prev_del = del;
      del = del->GetNext();
    }
    size_of_list = 0;
    std::cout << "Hexagon List deleted" << std::endl;
  }
}
```

titerator.h

```
#include <memory>
#ifndef INC_5_LAB__TITERATOR_H_
#define INC_5_LAB__TITERATOR_H_
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->GetValue(); }
  std::shared_ptr<T> operator->() { return node_ptr->GetValue(); }
  void operator++() { node_ptr = node_ptr->GetNext(); }
  Titerator operator++(int) {
    Titerator other(*this);
    ++(*this);
    return other;
  }
  bool operator==(Titerator const &i) { return node_ptr == i.node_ptr; };
  bool operator!=(Titerator const &i) { return node_ptr != i.node_ptr; };

private:
  std::shared_ptr<node> node_ptr;
};

#endif // INC_5_LAB__TITERATOR_H_
```

TVector.h

```
#ifndef DATA_VECTOR_H
#define DATA_VECTOR_H
```

```cpp
#include <iostream>

template<typename T>
class Vector {
public:
    Vector() {
        arr_ = new T[1];
        capacity_ = 1;
    }

    Vector(Vector &other) {
        if (this != &other) {
            delete[] arr_;
            arr_ = other.arr_;
            size_ = other.size_;
            capacity_ = other.capacity_;
            other.arr_ = nullptr;
            other.size_ = other.capacity_ = 0;
        }
    }

    Vector(Vector &&other) noexcept {
        if (this != &other) {
            delete[] arr_;
            arr_ = other.arr_;
            size_ = other.size_;
            capacity_ = other.capacity_;
            other.arr_ = nullptr;
            other.size_ = other.capacity_ = 0;
        }
    }

    Vector &operator=(Vector &other) {
        if (this != &other) {
            delete[] arr_;
            arr_ = other.arr_;
            size_ = other.size_;
            capacity_ = other.capacity_;
            other.arr_ = nullptr;
            other.size_ = other.capacity_ = 0;
        }
```

```
        return *this;
    }

    Vector &operator=(Vector &&other) noexcept {
        if (this != &other) {
            delete[] arr_;
            arr_ = other.arr_;
            size_ = other.size_;
            capacity_ = other.capacity_;
            other.arr_ = nullptr;
            other.size_ = other.capacity_ = 0;
        }
        return *this;
    }

    ~Vector() {
        delete[] arr_;
    }

public:
    [[nodiscard]] bool isEmpty() const {
        return size_ == 0;
    }

    [[nodiscard]] size_t size() const {
        return size_;
    }

    [[nodiscard]] size_t capacity() const {
        return capacity_;
    }

    void push_back(const T &value) {
        if (size_ >= capacity_) addMemory();
        arr_[size_++] = value;
    }

    void pop() {
        --size_;
    }

    T &back() {
```

```
        return arr_[size_ - 1];
    }

    void remove(size_t index) {
        for (size_t i = index + 1; i < size_; ++i) {
            arr_[i - 1] = arr_[i];
        }
        --size_;
    }

public:
    T *begin() {
        return &arr_[0];
    }

    const T *begin() const {
        return &arr_[0];
    }

    T *end() {
        return &arr_[size_];
    }

    const T *end() const {
        return &arr_[size_];
    }

public:
    T &operator[](size_t index) {
        return arr_[index];
    }

    const T &operator[](size_t index) const {
        return arr_[index];
    }

private:

    void addMemory() {
        capacity_ *= 2;
        T *tmp = arr_;
        arr_ = new T[capacity_];
```

```
        for (size_t i = 0; i < size_; ++i) arr_[i] = tmp[i];
        delete[] tmp;
    }

    T *arr_;
    size_t size_{};
    size_t capacity_{};
};

template<typename T>
inline std::ostream &operator<<(std::ostream &os, const Vector<T> &vec) {
    for (const T &val: vec) os << val << " ";
    return os;
}

#endif
```

TVector_item.h

```
#ifndef VECTOR_ITEM_H
#define VECTOR_ITEM_H

#include <memory>

template<typename T>
class Vector_item {
    public:
        Vector_item(): data(0) {};
        Vector_item(T t): data(t){};
        std::shared_ptr<Vector_item<T>> Get_next(){
            return next;
        };
        void Set_next(std::shared_ptr<Vector_item<T>> next_){
            next = next_;
        };
        T Get_data(){
            return data;
        }
    private:
        std::shared_ptr<Vector_item<T>> next = nullptr;
        T data;
};
```

```
#endif
```

tallocation_block.h

```
#ifndef TALLOCATION_BLOCK_H
#define TALLOCATION_BLOCK_H

#include "TVector.h"

class TAllocationBlock {
 public:
  TAllocationBlock(size_t size, size_t count);
  void* allocate();
  void deallocate(void* pointer);
  bool has_free_blocks();

  virtual ~TAllocationBlock();

 private:
  size_t _size;
  size_t _count;
  char* _used_blocks;
  Vector<void*> vec_free_blocks;
  size_t _free_count;
};

#endif  // TALLOCATION_BLOCK_H
```

main.cpp

```
#include <iostream>
#include "tlinkedlist.h"
#include "tallocation_block.h"
int main() {

  TLinkedList<Hexagon> tlinkedlist;
  std::cout << tlinkedlist.Empty() << std::endl;
  tlinkedlist.InsertLast(std::shared_ptr<Hexagon>(new Hexagon(Point(1,2),Point(2,3),Poin
  tlinkedlist.InsertLast(std::shared_ptr<Hexagon>(new Hexagon(Point(11,12),Point(12,13),
  tlinkedlist.InsertLast(std::shared_ptr<Hexagon>(new Hexagon(Point(17,18),Point(18,19),
  tlinkedlist.InsertLast(std::shared_ptr<Hexagon>(new Hexagon(Point(17,18),Point(18,19),
  std::cout << tlinkedlist;
  tlinkedlist.RemoveLast();
```

```
std::cout << tlinkedlist.Length() << std::endl;
tlinkedlist.RemoveFirst();
tlinkedlist.InsertFirst(std::shared_ptr<Hexagon>(new Hexagon(Point(2,3),Point(3,4),Poi
tlinkedlist.Insert(std::shared_ptr<Hexagon>(new Hexagon(Point(1,1),Point(2,3),Point(3,
std::cout << tlinkedlist.Empty() << std::endl;
std::cout << tlinkedlist.First() << std::endl;
std::cout << tlinkedlist.Last() << std::endl;
std::cout << tlinkedlist.GetItem(2) << std::endl;
tlinkedlist.Remove(2);
std::cout << tlinkedlist;
tlinkedlist.Clear();
TAllocationBlock allocator(sizeof(int), 10);
  int *a1 = nullptr;
  int *a2 = nullptr;
  int *a3 = nullptr;
  int *a4 = nullptr;
  int *a5 = nullptr;

a1 = (int *)allocator.allocate();
*a1 = 1;
std::cout << "a1 pointer value:" << *a1 << std::endl;

a2 = (int *) allocator.allocate();
*a2 = 2;
std::cout << "a2 pointer value:" << *a2 << std::endl;

a3 = (int *) allocator.allocate();
*a3 = 3;
std::cout << "a3 pointer value:" << *a3 << std::endl;

allocator.deallocate(a1);
allocator.deallocate(a3);

a4 = (int *) allocator.allocate();
*a4 = 4;
std::cout << "a4 pointer value:" << *a4 << std::endl;

a5 = (int *) allocator.allocate();
*a5 = 5;
std::cout << "a5 pointer value:" << *a5 << std::endl;

allocator.deallocate(a2);
```

```
  allocator.deallocate(a4);
  allocator.deallocate(a5);
  return 0;
}
```