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 *Фаттяхетдинов Сильвестр Динарович, 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:
Структура данных: список.

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 ликвидирован.

Недочёты

Недочётов не было обнаружено.

Выводы

Лабораторная работа №8 познакомила меня с понятием аллокатора. Аллокатор – довольно важная часть любой структуры данных, именно поэтому знакомство с аллокаторами – вещь обязательная для любого программиста.

Исходный код

```
#ifndef FIGURE_H
#define FIGURE_H
// #include <cstddef>
#include "point.h"

class Figure
{
public:
    virtual size_t VertexesNumber() = 0;
    virtual double Area() = 0;
    virtual void Print(std::ostream &os) = 0;
    ~Figure(){};
};

#endif // FIGURE_H
```

main.cpp

```
#include <iostream>
#include "rhombus.h"
#include "tnary_tree.h"
int main(){
    TNaryTree<Rhombus> t(3);
    //std::cout << "CrashTest";
    t.Update(std::shared_ptr<Rhombus>(new Rhombus(Point(0, 0), Point(0, 1),
Point(1, 1), Point(1, 0))), "");
    t.Update(std::shared_ptr<Rhombus>(new Rhombus(Point(0, 0), Point(0, 2),
Point(2, 2), Point(2, 0))), "c");
    t.Update(std::shared_ptr<Rhombus>(new Rhombus(Point(0, 0), Point(0, 3),
Point(3, 3), Point(3, 0))), "cb");
    t.Update(std::shared_ptr<Rhombus>(new Rhombus(Point(0, 0), Point(0, 7),
Point(7, 7), Point(7, 0))), "cbb");
    t.Update(std::shared_ptr<Rhombus>(new Rhombus(Point(0, 0), Point(0, 5),
Point(5, 5), Point(5, 0))), "cc");
    //std::cout << "CrashTest";
    TIterator<TnaryTreeItem<Rhombus>, Rhombus> iter(t.getroot());
    //std::cout << "CrashTest";
    std::cout << *iter << std::endl;
    iter.GoToSon();
    std::cout << *iter << std::endl;
    iter.GoToBro();
```

```
    std::cout << *iter << std::endl;
    TIterator<TnaryTreeItem<Rhombus>, Rhombus> iter1(t.getroot());
    TIterator<TnaryTreeItem<Rhombus>, Rhombus> iter2(t.getroot());
    if (iter == iter1) std::cout << "Iter = Iter1" << std::endl;
    if (iter !=iter1) std::cout << "Iter != Iter1" << std::endl;
    if (iter1 == iter2) std::cout << "Iter1 = Iter2" << std::endl;
    if (iter1 !=iter2) std::cout << "Iter1 != Iter2" << std::endl;
}
```

rhombus.cpp

```
#include "rhombus.h"
#include <cmath>

Rhombus::Rhombus() : a(0.0, 0.0), b(0.0, 0.0), c(0.0, 0.0), d(0.0, 0.0) {}

Rhombus::Rhombus(std::istream &is)
{
    is >> a >> b >> c >> d;
}

Rhombus::Rhombus(const Rhombus &other) : Rhombus(other.a, other.b, other.c,
other.d) {}

Rhombus::Rhombus(Point _a, Point _b, Point _c, Point _d)
{
    if (sqrt((_b.x() - _a.x()) * (_b.x() - _a.x()) +
             (_b.y() - _a.y()) * (_b.y() - _a.y())) ==
            sqrt((_c.x() - _b.x()) * (_c.x() - _b.x()) +
                 (_c.y() - _b.y()) * (_c.y() - _b.y())) &&
        sqrt((_c.x() - _b.x()) * (_c.x() - _b.x()) +
             (_c.y() - _b.y()) * (_c.y() - _b.y())) ==
            sqrt((_d.x() - _c.x()) * (_d.x() - _c.x()) +
                 (_d.y() - _c.y()) * (_d.y() - _c.y())) &&
        sqrt((_d.x() - _c.x()) * (_d.x() - _c.x()) +
             (_d.y() - _c.y()) * (_d.y() - _c.y())) ==
            sqrt((_a.x() - _d.x()) * (_a.x() - _d.x()) +
                 (_a.y() - _d.y()) * (_a.y() - _d.y()))) {
        a = _a;
        b = _b;
        c = _c;
        d = _d;
    } else {
        std::cout << "Invalid arguements";
```

```
    }
}

void Rhombus::Print(std::ostream &os)
{
    os << "Rhombus:";
    os << a << b << c << d << std::endl;
}

double Rhombus::Area()
{
    double s =
        abs(a.x() * b.y() + b.x() * c.y() + c.x() * d.y() + d.x() * a.y() -
            b.x() * a.y() - c.x() * b.y() - d.x() * c.y() - a.x() * d.y()) / 2;
    return s;
}

size_t Rhombus::VertexesNumber()
{
    return 4;
}

std::ostream &operator<<(std::ostream &os, const Rhombus &figure)
{
    os << "Rhombus: " << figure.a << " " << figure.b << " " << figure.c << " " <<
figure.d;
    return os;
}

Rhombus::~Rhombus() {}
```

rhombus.h

```
#ifndef RHOMBUS_H
#define RHOMBUS_H

#include "figure.h"

class Rhombus : public Figure
{
public:
    Rhombus();
    Rhombus(Point a, Point b, Point c, Point d);
    Rhombus(const Rhombus &other);
    Rhombus(std::istream &is);

    double Area();
    size_t VertexesNumber();
```

```
    void Print(std::ostream &os);

    friend std::ostream &operator<<(std::ostream &os, const Rhombus &figure);

    ~Rhombus();

private:
    Point a, b, c, d;
};

#endif // RHOMBUS_H
```

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) {}

Point::Point(std::istream &is)

{
    is >> x_ >> y_;
}

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

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

Point.h

```
#ifndef POINT_H
#define POINT_H

#include <iostream>

class Point
{
public:
    Point();
    Point(std::istream &is);
    Point(double x, double y);

    double x();
    double y();

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

private:
    double x_;
    double y_;
};

#endif // POINT_H
```

Tnary_tree.cpp

```
#ifndef TLIST_H
#define TLIST_H

#include "tnary_tree_item.h"
#include "titerator.h"

template <class T>
class TNaryTree
{
```

```
public:
    TNaryTree();
    TNaryTree(int n);

    void Update(std::shared_ptr<T> r, std::string tree_path);
    void RemoveSubTree(std::string tree_path);
    const std::shared_ptr<T> &GetItem(const std::string tree_path);
    bool Empty();
    std::shared_ptr<TnaryTreeItem<T>> getroot();
    template <class A>
    friend std::ostream &operator<<(std::ostream &os, const TNaryTree<A> &tree);

    virtual ~TNaryTree();

private:
    void Clearh(std::shared_ptr<TnaryTreeItem<T>> it);                        //
helper
    void Printh(std::shared_ptr<TnaryTreeItem<T>> it, std::ostream &os) const; //
helper
    int N;
    std::shared_ptr<TnaryTreeItem<T>> root;
};

#endif // TLIST_H
```

tnary_tree.h

```
#ifndef TLIST_H
#define TLIST_H

#include "tnary_tree_item.h"
#include "titerator.h"

template <class T>
class TNaryTree
{
public:
    TNaryTree();
    TNaryTree(int n);

    void Update(std::shared_ptr<T> r, std::string tree_path);
    void RemoveSubTree(std::string tree_path);
    const std::shared_ptr<T> &GetItem(const std::string tree_path);
    bool Empty();
    std::shared_ptr<TnaryTreeItem<T>> getroot();
    template <class A>
```

```
    friend std::ostream &operator<<(std::ostream &os, const TNaryTree<A> &tree);

    virtual ~TNaryTree();

private:
    void Clearh(std::shared_ptr<TnaryTreeItem<T>> it);                        //
helper
    void Printh(std::shared_ptr<TnaryTreeItem<T>> it, std::ostream &os) const; //
helper
    int N;
    std::shared_ptr<TnaryTreeItem<T>> root;
};

#endif // TLIST_H
```

tnary_tree_item.cpp

```
#include "tnary_tree_item.h"

template <class T>
TnaryTreeItem<T>::TnaryTreeItem(const std::shared_ptr<T> &r)
{
    this->rhombus = r;
    this->son = nullptr;
    this->brother = nullptr;
    std::cout << "Ntree item: created" << std::endl;
}

template <class T>
TnaryTreeItem<T>::TnaryTreeItem(const TnaryTreeItem &other)
{
    this->rhombus = other.rhombus;
    this->son = other.son;
    this->brother = other.brother;
    std::cout << "Ntree item: copied" << std::endl;
}

template <class T>
std::ostream &operator<<(std::ostream &os, const TnaryTreeItem<T> &obj)
{
    os << "Item: " << *obj.rhombus << std::endl;
    return os;
}
```

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

#include "rhombus.h"
template class TnaryTreeItem<Rhombus>;
template std::ostream& operator<<(std::ostream& os, const TnaryTreeItem<Rhombus>&
obj);
```

TIterator.h

```
#include <iostream>
#include <memory>

template <class item, class T>
class TIterator {
 public:
  TIterator(std::shared_ptr<item> n){
      node_ptr = n;
  }

 T operator*() { return *(node_ptr->rhombus); }

  // std::shared_ptr<T> operator->() { return node_ptr->GetValue(); }

 // void operator++() { node_ptr = node_ptr->GetNext(); }

  void GoToSon(){ //переход к сыну, если он есть
    if (node_ptr->son == nullptr){
      std::cout << "Node does not exist" << std::endl;
    } else {
      node_ptr = node_ptr->son;
    }
  }
  void GoToBro(){ //переход к брату, если он есть
    if (node_ptr->brother == nullptr){
      std::cout << "Node does not exist" << std::endl;
    } else {
      node_ptr = node_ptr->brother;
    }
  }

  bool operator==(TIterator const& i) { return node_ptr == i.node_ptr; }

  bool operator!=(TIterator const& i) { return !(*this == i); }
```

```
 private:
  std::shared_ptr<item> node_ptr;
};
```

tnary_tree_item.h

```
#ifndef TNARY_TREE_ITEM_H
#define TNARY_TREE_ITEM_H

#include <memory>
#include <iostream>

template <class T>
class TnaryTreeItem
{
public:
    // TnaryTreeItem();
    TnaryTreeItem(const std::shared_ptr<T> &r);
    TnaryTreeItem(const TnaryTreeItem &other);

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

    virtual ~TnaryTreeItem();

    std::shared_ptr<TnaryTreeItem<T>> son;
    std::shared_ptr<TnaryTreeItem<T>> brother;
    std::shared_ptr<T> rhombus;
};

#endif // TNARY_TREE_ITEM_H
```

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

HListItem.h

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

```
template <class T>
class HListItem
{
public:
    HListItem(const std::shared_ptr<Rhombus> &rhombus);
    template <class A>
    friend std::ostream &operator<<(std::ostream &os, HListItem<A> &obj);
    ~HListItem();
    std::shared_ptr<T> rhombus;
    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 << "rhombus 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<Rhombus> &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->rhombus;
}
template <class T>
std::shared_ptr<Rhombus> &TLinkedList<T>::First()
{
    return front->rhombus;
}
template <class T>
std::shared_ptr<Rhombus> &TLinkedList<T>::Last()
{
    return back->rhombus;
}
template <class T>
void TLinkedList<T>::InsertLast(const std::shared_ptr<Rhombus> &&rhombus)
{
    std::shared_ptr<HListItem<T>> obj(new HListItem<T>(Rhombus));
    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 << "rhombus does not pop_back, because the rhombus 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<Rhombus> &&rhombus)
{
    std::shared_ptr<HListItem<T>> obj(new HListItem<T>(rhombus));
    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 << "rhombus does not pop_front, because the rhombus 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<Rhombus> &&rhombus, 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>(rhombus));
        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 rhombus list is empty, so there is nothing to output" <<
std::endl;
    }
    else
    {
        os << "Print rhombus List" << std::endl;
        std::shared_ptr<HListItem<T>> obj = hl.front;
```

```
        while (obj != nullptr)
        {
            if (obj->next != nullptr)
            {
                os << obj->rhombus << " "
                   << ","
                   << " ";
                obj = obj->next;
            }
            else
            {
                os << obj->rhombus;
                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 << "rhombus List deleted" << std::endl;
    }
}
```

TLinkedList.h

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

template <class T>
class TLinkedList
```

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