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:  
  
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 и дополнительного контейнера, позже они были полностью ликвидирован.

**Недочёты**  
Недочётов не было обнаружено.

**Выводы**

Лабораторная работа №8 позволила мне реализовать свой аллокатор, полностью прочувствовать процесс выделения памяти на низкоуровневых языках программирования. Также данная лабораторная работа помогла осознать, что необходимо уделять особое внимание памяти и обращением с ней.

**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<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 "rhombus.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 "rhombus.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