МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ  
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСТИТЕТ)

**ЛАБОРАТОРНАЯ РАБОТА №8**

по курсу “Объектно-ориентированное программирование”

I семестр, 2021/22 учебный год

Студент: *Попов Андрей Викторович, группа М8О-207Б-20*

Преподаватель: *Дорохов Евгений Павлович, каф. 806*

**Задание:**

Используя структуру данных, разработанную для лабораторной работы №7, спроектировать и разработать аллокатор памяти для динамической структуры данных. Целью построения аллокатора является 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 операторы new и delete у классов-фигур.

**Вариант №20:**

* + Фигура: Трапеция(Trapezoid)
  + Контейнер первого уровня: Н-дерево (TNaryTree)
  + Контейнер второго уровня: Связный список (TLinkedList)

**Описание программы:**

Исходный код разделён на 17 файлов:

* figure.h – описание класса фигуры
* point.h – описание класса точки
* point.cpp – реализация класса точки
* trapezoid.h – описание класса трапеции (наследуется от фигуры)
* trapezoid.cpp – реализация класса трапеции
* TNaryTree\_Item.h – описание элемента дерева
* TNaryTree\_Item.cpp – реализация элемента дерева
* TNaryTree.h – описание дерева
* TNaryTree.cpp – реализация дерева
* Iterator.h – реализация итератора
* TAllocatorBlock.h/cpp – реализация класса аллокатора для фигуры
* TLinkedList.h/cpp – реализация класса связного списка для использования в аллокаторе
* TLinkedListItem.h/cpp – реализация класса элемента связного списка для использования в аллокаторе
* main.cpp – основная программа

**Дневник отладки:**

При выполнении работы ошибок выявлено не было.

**Вывод:**  
 В процессе выполнения работы я на практике познакомился с понятием аллокатора. Написание собственноручного аллокатора помогает реализовать собственную логику выделения памяти, которая может быть более оправданной в некоторых ситуациях, чем стандартный аллокатор, как для самописных, так и для стандартных структур данных.

**Исходный код:**

**figure.h**

**#ifndef FIGURE\_H**

**#define FIGURE\_H**

**#include "point.h"**

**class Figure {**

**public:**

**virtual size\_t VertexesNumber() = 0;**

**virtual double Area() = 0;**

**virtual void Print(std::ostream& os) = 0;**

**~Figure() {};**

**};**

**#endif**

**Iterator.h**

**#ifndef ITERATOR\_H**

**#define ITERATOR\_H**

**#include "TNaryTree.h"**

**#include <iostream>**

**template <class node, class T>**

**class Iterator{**

**public:**

**Iterator(std::shared\_ptr<node> n){**

**it = n;**

**it\_prev = nullptr;**

**}**

**Iterator& operator=(const Iterator& it\_){**

**it = it\_.it;**

**return \*this;**

**}**

**bool operator==(const Iterator& it\_){**

**return it == it\_.it;**

**}**

**bool operator!=(const Iterator& it\_){**

**return !(it == it\_.it);**

**}**

**node operator\*(){**

**return it->Get\_data();**

**}**

**Iterator& operator++ (){**

**if(it->Get\_son() != nullptr && it\_prev != it->Get\_bro()){**

**it\_prev = it;**

**it = it->Get\_son();**

**} else if (it->Get\_bro() != nullptr && it\_prev != it->Get\_bro()){**

**it\_prev = it;**

**it = it->Get\_bro();**

**} else {**

**while(it->Get\_bro() == nullptr || (it->Get\_bro() == it\_prev)){**

**it\_prev = it;**

**if(it->Get\_older() == nullptr){**

**return \*this;**

**}**

**it = it->Get\_older();**

**}**

**if(it->Get\_bro() == nullptr && it->Get\_son() == it\_prev){**

**it\_prev = it;**

**it = it->Get\_older();**

**if(it->Get\_older() == nullptr){**

**return \*this;**

**}**

**if(it->Get\_bro() == nullptr){**

**while(it->Get\_bro() == nullptr){**

**it\_prev = it;**

**it = it->Get\_older();**

**}**

**it\_prev = it;**

**it = it->Get\_bro();**

**} else {**

**it = it->Get\_bro();**

**}**

**} else {**

**it = it->Get\_bro();**

**}**

**}**

**return \*this;**

**}**

**Iterator& operator++ (int){**

**Iterator tmp(\*it);**

**++(\*this);**

**return it;**

**}**

**Iterator& operator-- (){**

**it = it.Get\_older();**

**return \*this;**

**}**

**Iterator& operator-- (int){**

**Iterator tmp(\*it);**

**--(\*this);**

**return it;**

**}**

**public:**

**std::shared\_ptr<node> it\_prev;**

**std::shared\_ptr<node> it;**

**};**

**#endif**

**main.cpp**

**#include "TNaryTree.h"**

**#include "trapezoid.h"**

**#include "Iterator.h"**

**#include <unistd.h>**

**int main(void){**

**TNaryTree<Trapezoid> t(9);**

**t.Update(Trapezoid(Point(0, 0), Point(1, 0),Point(1, 1), Point(0, 1)), "");**

**t.Update(Trapezoid(Point(0, 0), Point(6, 0),Point(6, 1), Point(0, 1)), "c");**

**t.Update(Trapezoid(Point(0, 0), Point(3, 0),Point(3, 1), Point(0, 1)), "cc");**

**t.Update(Trapezoid(Point(0, 0), Point(2, 0),Point(2, 1), Point(0, 1)), "cb");**

**t.Update(Trapezoid(Point(0, 0), Point(4, 0),Point(4, 1), Point(0, 1)), "cbc");**

**t.Update(Trapezoid(Point(0, 0), Point(9, 0),Point(9, 1), Point(0, 1)), "cbcc");**

**t.Update(Trapezoid(Point(0, 0), Point(5, 0),Point(5, 1), Point(0, 1)), "b");**

**t.Update(Trapezoid(Point(0, 0), Point(7, 0),Point(7, 1), Point(0, 1)), "cbcb");**

**t.Update(Trapezoid(Point(0, 0), Point(8, 0),Point(8, 1), Point(0, 1)), "cbcbb");**

**std::cout << \*(t.Tree\_root()->Get\_bro());**

**std::cout << t.size() << "\n";**

**std::cout << t.Area("") << "\n";**

**std::cout << t.size() << "\n";**

**std::cout << t;**

**TNaryTree<Trapezoid> q(t);**

**t.Clear();**

**std::cout << q.size() << " " << q.Area("") << "\n";**

**std::cout << q;**

**//std::cout << q.Tree\_root()->Get\_data() << "\n";**

**for(auto i: q){**

**std::cout << i.Area() << " ";**

**//sleep(1);**

**}**

**std::cout << "\n";**

**//std::cout << (q.Tree\_root()->Get\_older()->Get\_bro());**

**return 0;**

**}**

**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);**

**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**

**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::dist(Point& other) {**

**double dx = (other.x\_ - x\_);**

**double dy = (other.y\_ - y\_);**

**return std::sqrt(dx\*dx + dy\*dy);**

**}**

**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;**

**}**

**TAllocationBlock.h**

**#ifndef LAB6\_TALLOCATIONBLOCK\_H**

**#define LAB6\_TALLOCATIONBLOCK\_H**

**#include <iostream>**

**#include <cstdlib>**

**#include "TLinkedList.h"**

**class TAllocationBlock {**

**public:**

**TAllocationBlock(int32\_t size, int32\_t count);**

**void \*Allocate();**

**void Deallocate(void \*ptr);**

**bool Empty();**

**int32\_t Size();**

**virtual ~TAllocationBlock();**

**private:**

**char \*used\_bl;**

**TLinkedList free\_bl;**

**};**

**#endif //LAB6\_TALLOCATIONBLOCK\_H**

**TAllocationBlock.cpp**

**#include "TAllocationBlock.h"**

**TAllocationBlock::TAllocationBlock(int32\_t size, int32\_t count) {**

**used\_bl = (char \*)malloc(size \* count);**

**for (int32\_t i = 0; i < count; ++i) {**

**void \*ptr = (void \*)malloc(sizeof(void \*));**

**ptr = used\_bl + i \* size;**

**free\_bl.InsertLast(ptr);**

**}**

**}**

**void \*TAllocationBlock::Allocate() {**

**if (!free\_bl.Empty()) {**

**void \*res = free\_bl.GetBlock();**

**int32\_t first = 1;**

**free\_bl.Remove(first);**

**std::cout << "Rectangle created" << std::endl;**

**return res;**

**} else {**

**throw std::bad\_alloc();**

**}**

**}**

**void TAllocationBlock::Deallocate(void \*ptr) {**

**free\_bl.InsertFirst(ptr);**

**}**

**bool TAllocationBlock::Empty() {**

**return free\_bl.Empty();**

**}**

**int32\_t TAllocationBlock::Size() {**

**return free\_bl.Length();**

**}**

**TAllocationBlock::~TAllocationBlock() {**

**while (!free\_bl.Empty()) {**

**int32\_t first = 1;**

**free\_bl.Remove(first);**

**}**

**free(used\_bl);**

**std::cout << "Rectangle deleted" << std::endl;**

**}**

**TLinkedList.h**

**#ifndef LAB6\_TLINKEDLIST\_H**

**#define LAB6\_TLINKEDLIST\_H**

**#include "TLinkedListItem.h"**

**#include <memory>**

**#include <iostream>**

**class TLinkedList {**

**public:**

**TLinkedList();**

**void InsertFirst(void \*link);**

**void InsertLast(void \*link);**

**void Insert(int position, void \*link);**

**int Length();**

**bool Empty();**

**void Remove(int &position);**

**void Clear();**

**void\* GetBlock();**

**virtual ~TLinkedList();**

**private:**

**TLinkedListItem\* first;**

**};**

**#endif // LAB6\_TLINKEDLIST\_H**

**TLinkedList.cpp**

**#include "TLinkedList.h"**

**TLinkedList::TLinkedList() {**

**first = nullptr;**

**}**

**void TLinkedList::InsertFirst(void\* link) {**

**auto \*other = new TLinkedListItem(link);**

**other->SetNext(first);**

**first = other;**

**}**

**void TLinkedList::Insert(int position, void \*link) {**

**TLinkedListItem \*iter = this->first;**

**auto \*other = new TLinkedListItem(link);**

**if (position == 1) {**

**other->SetNext(iter);**

**this->first = other;**

**} else {**

**if (position <= this->Length()) {**

**for (int i = 1; i < position - 1; ++i)**

**iter = iter->GetNext();**

**other->SetNext(iter->GetNext());**

**iter->SetNext(other);**

**}**

**}**

**}**

**void TLinkedList::InsertLast(void \*link) {**

**auto \*other = new TLinkedListItem(link);**

**TLinkedListItem \*iter = this->first;**

**if (first != nullptr) {**

**while (iter->GetNext() != nullptr) {**

**iter = iter->SetNext(iter->GetNext());**

**}**

**iter->SetNext(other);**

**other->SetNext(nullptr);**

**}**

**else {**

**first = other;**

**}**

**}**

**int TLinkedList::Length() {**

**int len = 0;**

**TLinkedListItem\* item = this->first;**

**while (item != nullptr) {**

**item = item->GetNext();**

**len++;**

**}**

**return len;**

**}**

**bool TLinkedList::Empty() {**

**return first == nullptr;**

**}**

**void TLinkedList::Remove(int &position) {**

**TLinkedListItem \*iter = this->first;**

**if (position <= this->Length()) {**

**if (position == 1) {**

**this->first = iter->GetNext();**

**} else {**

**int i = 1;**

**for (i = 1; i < position - 1; ++i) {**

**iter = iter->GetNext();**

**}**

**iter->SetNext(iter->GetNext()->GetNext());**

**}**

**} else {**

**std::cout << "error" << std::endl;**

**}**

**}**

**void TLinkedList::Clear() {**

**first = nullptr;**

**}**

**void \* TLinkedList::GetBlock() {**

**return this->first->GetBlock();**

**}**

**TLinkedList::~TLinkedList() {**

**delete first;**

**}**

**TLinkedListItem.h**

**#ifndef LAB6\_TLINKEDLISTITEM\_H**

**#define LAB6\_TLINKEDLISTITEM\_H**

**#include <memory>**

**class TLinkedListItem {**

**public:**

**TLinkedListItem(void \*link);**

**TLinkedListItem\* SetNext(TLinkedListItem\* next);**

**TLinkedListItem\* GetNext();**

**void\* GetBlock();**

**virtual ~TLinkedListItem();**

**private:**

**void\* link;**

**TLinkedListItem\* next;**

**};**

**#endif // LAB6\_TLINKEDLISTITEM\_H**

**TLinkedListItem.cpp**

**#include "TLinkedListItem.h"**

**#include <iostream>**

**TLinkedListItem::TLinkedListItem(void\* link) {**

**this->link = link;**

**this->next = nullptr;**

**}**

**TLinkedListItem\* TLinkedListItem::SetNext(TLinkedListItem\* next) {**

**TLinkedListItem\* old = this->next;**

**this->next = next;**

**return old;**

**}**

**TLinkedListItem\* TLinkedListItem::GetNext() {**

**return this->next;**

**}**

**void\* TLinkedListItem::GetBlock() {**

**return this->link;**

**}**

**TLinkedListItem::~TLinkedListItem() {**

**}**

**TNaryTree.h**

**#ifndef TNARYTREE\_H**

**#define TNARYTREE\_H**

**#include "TNaryTree\_item.h"**

**#include "Iterator.h"**

**template <class T>**

**class TNaryTree {**

**public:**

**// Инициализация дерева с указанием размера**

**TNaryTree(int n);**

**// Полное копирование дерева**

**TNaryTree(const TNaryTree<T>& other);**

**// Добавление или обновление вершины в дереве согласно заданному пути.**

**// Путь задается строкой вида: "cbccbccc",**

**// где 'c' - старший ребенок, 'b' - младший брат**

**// последний символ строки - вершина, которую нужно добавить или обновить.**

**// Пустой путь "" означает добавление/обновление корня дерева.**

**// Если какой-то вершины в tree\_path не существует,**

**// то функция должна бросить исключение std::invalid\_argument**

**// Если вершину нельзя добавить из за переполнения,**

**// то функция должна бросить исключение std::out\_of\_range**

**void Update(T &&polygon, std::string &&tree\_path = "");**

**// Удаление поддерева**

**void Clear(std::string &&tree\_path = "");**

**// Проверка наличия в дереве вершин**

**bool Empty();**

**// Подсчет суммарной площади поддерева**

**double Area(std::string &&tree\_path);**

**int size();**

**// Вывод дерева в формате вложенных списков, где каждый вложенный список является:**

**// "S0: [S1: [S3, S4: [S5, S6]], S2]", где Si - площадь фигуры**

**std::shared\_ptr<Item<T>> Tree\_root();**

**std::shared\_ptr<Item<T>> Tree\_end();**

**template <class A>**

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

**Iterator<Item<T>, T> begin();**

**Iterator<Item<T>, T> end();**

**virtual ~TNaryTree();**

**private:**

**int curr\_number;**

**int max\_number;**

**std::shared\_ptr<Item<T>> root;**

**std::shared\_ptr<Item<T>> fin;**

**};**

**#endif**

**TNaryTree.cpp**

**#include "TNaryTree.h"**

**#include "Iterator.h"**

**#include "point.h"**

**#include <string>**

**#include <memory>**

**#include <stdexcept>**

**#include <iostream>**

**template <class T>**

**TNaryTree<T>::TNaryTree(int n) {**

**max\_number = n;**

**curr\_number = 0;**

**root = nullptr;**

**fin = std::make\_shared<Item<T>>(T());**

**};**

**template <class T>**

**std::shared\_ptr<Item<T>> TNaryTree<T>::Tree\_root(){**

**return root;**

**}**

**template <class T>**

**std::shared\_ptr<Item<T>> TNaryTree<T>::Tree\_end(){**

**std::shared\_ptr<Item<T>> tmp(new(Item<T>));**

**while((\*tmp).Get\_son() != nullptr){**

**tmp = (\*tmp).Get\_son();**

**}**

**return tmp;**

**}**

**template <class T>**

**bool TNaryTree<T>::Empty() {**

**return curr\_number ? 0 : 1;**

**}**

**template <class T>**

**void TNaryTree<T>::Update(T &&polygon, std::string &&tree\_path){**

**std::cout << tree\_path.length() << " " << tree\_path << "\n";**

**if(tree\_path != "" && curr\_number == 0){**

**throw std::invalid\_argument("Error, there is not a root value\n");**

**return;**

**} else if(tree\_path == "" && curr\_number == 0){**

**std::shared\_ptr<Item<T>> q(new Item<T>(polygon));**

**(\*q).Set\_older(fin);**

**root = q;**

**curr\_number++;**

**} else if(curr\_number + 1 > max\_number){**

**throw std::out\_of\_range("Current number of elements equals maximal number of elements in tree\n");**

**return;**

**} else {**

**std::shared\_ptr<Item<T>> tmp = root;**

**for(size\_t i = 0; i < tree\_path.length() - 1; i++) {**

**if(tree\_path[i] == 'b'){**

**std::shared\_ptr<Item<T>> q((\*tmp).Get\_bro());**

**if(q == nullptr){**

**throw std::invalid\_argument("Path does not exist\n");**

**return;**

**}**

**tmp = q;**

**} else if(tree\_path[i] == 'c'){**

**std::shared\_ptr<Item<T>> q = (\*tmp).Get\_son();**

**if(q == nullptr){**

**throw std::invalid\_argument("Path does not exist\n");**

**return;**

**}**

**tmp = q;**

**} else {**

**std::cout << tree\_path.length() << " " << tree\_path << "\n";**

**throw std::invalid\_argument("Error in path\n");**

**return;**

**}**

**}**

**std::shared\_ptr<Item<T>> item(new Item<T>(polygon));**

**if(tree\_path.back() == 'b'){**

**/\*std::shared\_ptr<Item> p = (\*tmp).Get\_bro();**

**p = item;\*/**

**(\*item).Set\_older(tmp);**

**(\*tmp).Set\_bro(item);**

**curr\_number++;**

**} else if(tree\_path.back() == 'c'){**

**/\*std::shared\_ptr<Item> p = (\*tmp).Get\_son();**

**p = item;\*/**

**(\*item).Set\_older(tmp);**

**(\*tmp).Set\_son(item);**

**curr\_number++;**

**} else {**

**throw std::invalid\_argument("Error in path\n");**

**return;**

**}**

**}**

**}**

**template <class T>**

**std::shared\_ptr<Item<T>> copy(std::shared\_ptr<Item<T>> root){**

**if(!root){**

**return nullptr;**

**}**

**std::shared\_ptr<Item<T>> root\_copy(new Item<T>(root));**

**(\*root\_copy).Set\_older((\*root).Get\_older());**

**(\*root\_copy).Set\_bro(copy((\*root).Get\_bro()));**

**(\*root\_copy).Set\_son(copy((\*root).Get\_son()));**

**return root\_copy;**

**}**

**template <class T>**

**TNaryTree<T>::TNaryTree(const TNaryTree<T>& other){**

**curr\_number = 0;**

**max\_number = other.max\_number;**

**root = copy(other.root);**

**curr\_number = other.curr\_number;**

**;}**

**template <class T>**

**int TNaryTree<T>::size(){**

**return curr\_number;**

**}**

**template <class T>**

**int clear(std::shared\_ptr<Item<T>> node) {**

**if (!node) {**

**return 0;**

**}**

**int temp\_res = clear((\*node).Get\_bro()) + clear((\*node).Get\_son()) + 1;**

**return temp\_res;**

**}**

**template <class T>**

**void TNaryTree<T>::Clear(std::string &&tree\_path){**

**std::shared\_ptr<Item<T>> prev\_tmp = nullptr;**

**std::shared\_ptr<Item<T>> tmp;**

**tmp = root;**

**if (tree\_path.empty()) {**

**clear(root);**

**curr\_number = 0;**

**root = nullptr;**

**return;**

**}**

**for(size\_t i = 0; i < tree\_path.length(); i++) {**

**if(tree\_path[i] == 'b'){**

**std::shared\_ptr<Item<T>> q((\*tmp).Get\_bro());**

**if(q == nullptr){**

**throw std::invalid\_argument("Path does not exist\n");**

**return;**

**}**

**prev\_tmp = tmp;**

**tmp = q;**

**} else if(tree\_path[i] == 'c'){**

**std::shared\_ptr<Item<T>> q((\*tmp).Get\_son());**

**if(q == nullptr){**

**throw std::invalid\_argument("Path does not exist\n");**

**return;**

**}**

**prev\_tmp = tmp;**

**tmp = q;**

**} else {**

**throw std::invalid\_argument("Error in path\n");**

**return;**

**}**

**}**

**if (tmp == (\*prev\_tmp).Get\_son()) {**

**(\*prev\_tmp).Set\_son(nullptr);**

**} else {**

**(\*prev\_tmp).Set\_bro(nullptr);**

**}**

**curr\_number -= clear(tmp);**

**}**

**template <class T>**

**double area(std::shared\_ptr<Item<T>> node){**

**if(!node){**

**return 0;**

**}**

**return node->Area() + area((\*node).Get\_bro()) + area((\*node).Get\_son());**

**}**

**template <class T>**

**double TNaryTree<T>::Area(std::string &&tree\_path){**

**std::shared\_ptr<Item<T>> tmp;**

**tmp = root;**

**for(size\_t i = 0; i < tree\_path.length(); i++) {**

**if(tree\_path[i] == 'b'){**

**std::shared\_ptr<Item<T>> q((\*tmp).Get\_bro());**

**if(q == nullptr){**

**throw std::invalid\_argument("Path does not exist\n");**

**return -1;**

**}**

**tmp = q;**

**} else if(tree\_path[i] == 'c'){**

**std::shared\_ptr<Item<T>> q((\*tmp).Get\_son());**

**if(q == nullptr){**

**throw std::invalid\_argument("Path does not exist\n");**

**return -1;**

**}**

**tmp = q;**

**} else {**

**throw std::invalid\_argument("Error in path\n");**

**return -1;**

**}**

**}**

**return area(tmp);**

**}**

**// Вывод дерева в формате вложенных списков, где каждый вложенный список является:**

**// "S0: [S1: [S3, S4: [S5, S6]], S2]", где Si - площадь фигуры**

**template <class T>**

**void print(std::ostream& os, std::shared\_ptr<Item<T>> node){**

**if(!node){**

**return;**

**}**

**if((\*node).Get\_son()){**

**//os << <<node->pentagon.GetArea() << : ]" <<**

**os << node->Area() << ": [";**

**print(os, (\*node).Get\_son());**

**os << "]";**

**if((\*node).Get\_bro()){**

**if((\*node).Get\_bro()){**

**os << ", ";**

**print(os, (\*node).Get\_bro());**

**}**

**}**

**} else if ((\*node).Get\_bro()) {**

**os << node->Area() << ", ";**

**print(os, (\*node).Get\_bro());**

**if((\*node).Get\_son()){**

**os << ": [";**

**print(os, (\*node).Get\_son());**

**os << "]";**

**}**

**}**

**else {**

**os << node->Area();**

**}**

**}**

**template <class T>**

**std::ostream& operator<<(std::ostream& os, const TNaryTree<T>& tree){**

**print(os, tree.root);**

**os << "\n";**

**return os;**

**}**

**template <class T>**

**Iterator<Item<T>, T> TNaryTree<T>::begin(){**

**return Iterator<Item<T>, T>(root);**

**}**

**template <class T>**

**Iterator<Item<T>, T> TNaryTree<T>::end(){**

**return Iterator<Item<T>, T>(root->Get\_older());**

**}**

**template <class T>**

**TNaryTree<T>::~TNaryTree(){**

**Clear();**

**};**

**#include "trapezoid.h"**

**template class TNaryTree<Trapezoid>;**

**template std::ostream& operator<<(std::ostream& os, const TNaryTree<Trapezoid>& stack);**

**TNaryTree\_item.h**

**#ifndef TNARYTREE\_ITEM\_H**

**#define TNARYTREE\_ITEM\_H**

**#include <memory>**

**#include "trapezoid.h"**

**template <class T>**

**class Item {**

**public:**

**Item(T a);**

**Item(std::shared\_ptr<Item<T>> a);**

**Item();**

**void Set(T a);**

**void Set\_bro(std::shared\_ptr<Item<T>> bro\_);**

**void Set\_son(std::shared\_ptr<Item<T>> son\_);**

**void Set\_older(std::shared\_ptr<Item<T>> older\_);**

**T Get\_data();**

**std::shared\_ptr<Item<T>> Get\_bro();**

**std::shared\_ptr<Item<T>> Get\_son();**

**std::shared\_ptr<Item<T>> Get\_older();**

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

**double Area();**

**~Item();**

**template<class A>**

**friend std::ostream &operator<<(std::ostream &os, const Item<A> &obj);**

**private:**

**std::shared\_ptr<Item<T>> bro = nullptr;**

**std::shared\_ptr<Item<T>> son = nullptr;**

**std::shared\_ptr<Item<T>> older = nullptr;**

**T data;**

**};**

**#endif**

**TNaryTree\_item.cpp**

**#include "TNaryTree\_item.h"**

**#include <iostream>**

**template <class T>**

**Item<T>::Item() {**

**data = T();**

**}**

**template <class T>**

**Item<T>::Item(T a){**

**data = a;**

**}**

**template <class T>**

**void Item<T>::Set(T a){**

**data = a;**

**}**

**template <class T>**

**T Item<T>::Get\_data(){**

**return data;**

**}**

**template <class T>**

**std::shared\_ptr<Item<T>> Item<T>::Get\_bro(){**

**return bro;**

**}**

**template <class T>**

**std::shared\_ptr<Item<T>> Item<T>::Get\_son(){**

**return son;**

**}**

**template <class T>**

**std::shared\_ptr<Item<T>> Item<T>::Get\_older(){**

**return this == nullptr ? nullptr : older;**

**}**

**template <class T>**

**Item<T>::Item(std::shared\_ptr<Item<T>> a){**

**older = nullptr;**

**bro = a->bro;**

**son = a->son;**

**data = a->data;**

**}**

**template <class T>**

**void Item<T>::Print(std::ostream &os){**

**os << data.Area();**

**}**

**template <class T>**

**void Item<T>::Set\_bro(std::shared\_ptr<Item<T>> bro\_){**

**bro = bro\_;**

**}**

**template <class T>**

**void Item<T>::Set\_son(std::shared\_ptr<Item<T>> son\_){**

**son = son\_;**

**}**

**template <class T>**

**void Item<T>::Set\_older(std::shared\_ptr<Item<T>> older\_){**

**older = older\_;**

**}**

**template <class T>**

**double Item<T>::Area(){**

**return data.Area();**

**}**

**template <class T>**

**std::ostream &operator<<(std::ostream &os, const Item<T> &obj){**

**return os << "Item: " << obj.data << std::endl;**

**}**

**template <class T>**

**Item<T>::~Item() {};**

**#include "trapezoid.h"**

**template class Item<Trapezoid>;**

**template std::ostream& operator<<(std::ostream& os, const Item<Trapezoid> &obj);**

**trapezoid.h**

**#ifndef TRAPEZOID\_H**

**#define TRAPEZOID\_H**

**#include "figure.h"**

**class Trapezoid: Figure {**

**public:**

**size\_t VertexesNumber();**

**double Area();**

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

**Trapezoid();**

**Trapezoid(Point a\_, Point b\_, Point c\_, Point d\_);**

**Trapezoid(std::istream& is);**

**friend std::istream &operator>>(std::istream &is, Trapezoid &figure);**

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

**void\* operator new(size\_t size);**

**void operator delete(void\* ptr);**

**~Trapezoid();**

**private:**

**static TAllocatorBlock tblock;**

**Point a;**

**Point b;**

**Point c;**

**Point d;**

**};**

**#endif**

**trapezoid.cpp**

**#include <cmath>**

**#include "point.h"**

**#include "trapezoid.h"**

**double Trapezoid::Area(){**

**double la = a.dist(d);**

**double lb = b.dist(c);**

**double lc = c.dist(d);**

**double ld = a.dist(b);**

**if (la > lb) {**

**double t = la;**

**la = lb;**

**lb = t;**

**}**

**double numenator = (lb - la) \* (lb - la) + lc \* lc - ld \* ld;**

**double denumenator = 2 \* (lb - la);**

**if (denumenator == 0) {**

**return (la \* lc);**

**}**

**double h = sqrt(lc \* lc - ((numenator \* numenator) / (denumenator \* denumenator)));**

**return ((la + lb) / 2 \* h);**

**}**

**void Trapezoid::Print(std::ostream& os){**

**os << a << " " << b << " " << c << " " << d << "\n";**

**}**

**size\_t Trapezoid::VertexesNumber(){**

**return (size\_t)(4);**

**}**

**Trapezoid::Trapezoid() : a(Point()), b(Point()), c(Point()), d(Point()){**

**}**

**Trapezoid::Trapezoid(Point a\_, Point b\_, Point c\_, Point d\_):**

**a(a\_), b(b\_), c(c\_), d(d\_){**

**}**

**Trapezoid::Trapezoid(std::istream& is){**

**is >> a >> b >> c >> d;**

**}**

**std::istream &operator>>(std::istream &is, Trapezoid &figure){**

**is >> figure.a >> figure.b >> figure.c >> figure.d;**

**return is;**

**}**

**std::ostream &operator<<(std::ostream &os, const Trapezoid &figure){**

**os << "Trapezoid: " << figure.a << " " << figure.b << " " << figure.c << " " << figure.d << std::endl;**

**return os;**

**}**

**void\* Trapezoid::operator new(size\_t size){**

**return tblock.Allocate(size);**

**}**

**void Trapezoid::operator delete(void\* ptr){**

**tblock.Deallocate(ptr);**

**}**

**Trapezoid::~Trapezoid() {}**

**Результат работы:**

5

17

5

5 17

1: [4], 4, 4: [4]

1: [4], 4, 4: [4]

0