Московский Авиационный Институт

(Национальный Исследовательский Университет)

**Лабораторная работа №8 по курсу**

**«Объектно-ориентированное программирование»**

Студент: Шаларь Игорь Павлович

Группа: М8О-208Б-20

Вариант: 23

Преподаватель: Дорохов Евгений Павлович

Оценка: \_\_\_\_\_\_\_\_\_\_\_

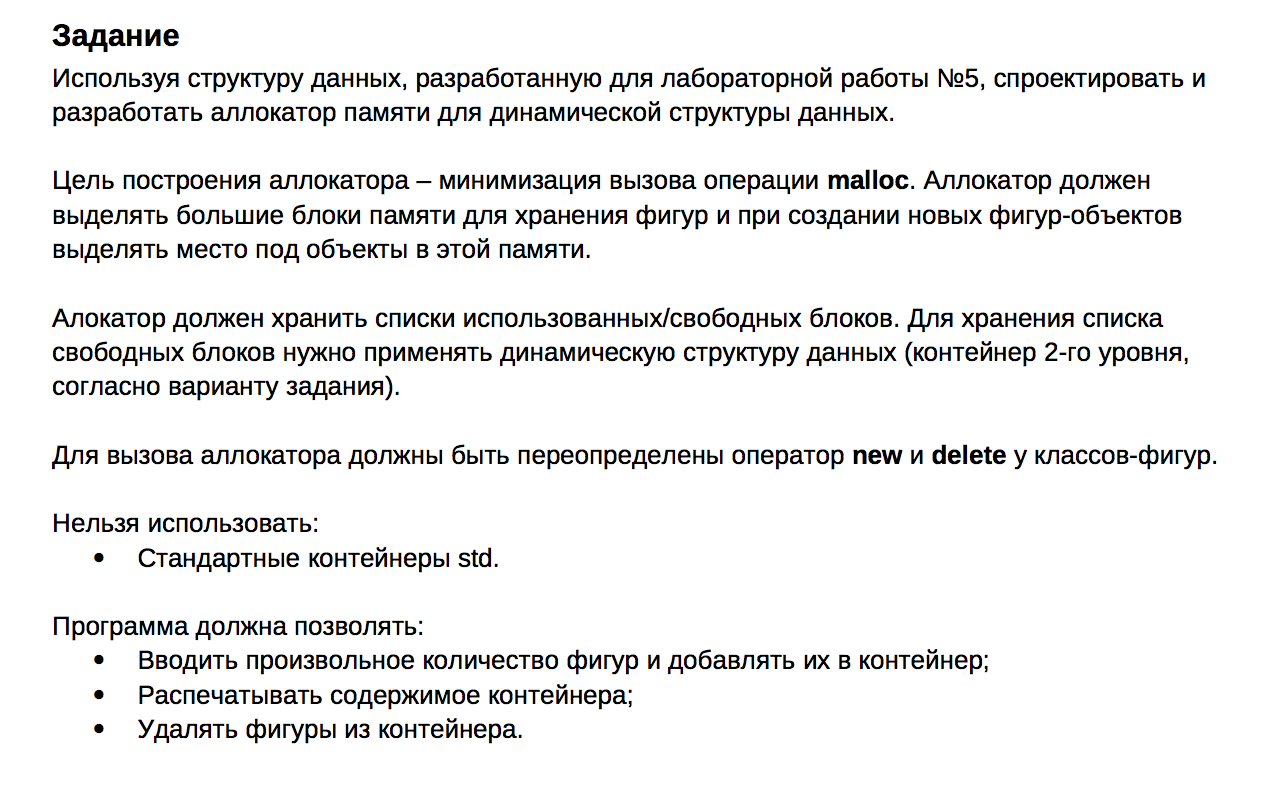
Дата: \_\_\_\_\_\_\_\_\_\_\_

**Цель:**

* -Закрепление навыков по работе с памятью в C++;
* -Создание аллокаторов памяти для динамических структур данных.

**Задание:**

Вариант 23: hexagon tnarytree tqueue



**Описание программы:**

figure.h - описание родительского класса для класса-фигуры.

hexagon.h - заголовочный файл описывающий классы.

tnarytree.h - реализация структуры.

tqueue.h - реализация очереди.

hexagon.cpp - реализация.

main.cpp - основной файл, взаимодействие с пользователем.

**Пример работы:**

USAGE:

Add element:

a [x1] [y1] [x2] [y2] [x3] [y3] [x4] [y4] [x5] [y5] [x6] [y6]

[route]

example:

a 0 2 1 1 1 -1 0 -2 -1 -1 -1 1

cbb

Print: p

Delete:

d

[path]

Area:

s

[path]

Traversal with iterator: i

Stop: q

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Enter tree size:

100

a 0 2 1 1 1 -1 0 -2 -1 -1 -1 1

added

a 0 0 0 0 0 0 0 0 0 0 0 0

c

added

p

6: [0]

a 0 2 1 1 1 -1 0 -2 -1 -1 -1 1

cb

added

i

Hexagon: (0, 2) (1, 1) (1, -1) (0, -2) (-1, -1) (-1, 1)

Hexagon: (0, 0) (0, 0) (0, 0) (0, 0) (0, 0) (0, 0)

Hexagon: (0, 2) (1, 1) (1, -1) (0, -2) (-1, -1) (-1, 1)

s

cb

6

d

deleted

i

q

**Дневник отладки:**

Очередь работала не совсем корректно. Перегрузить операторы new и delete корректно получилось не сразу.

**Выводы:**

Научился реализовывать аллокаторы памяти для динамических структур данных. Если при работе с небольшим объемом данных их преимущества незаметны, то при обработке большого объема данных аллокатор значительно повысил бы эффективность структуры. Также я научился перегружать операторы new и delete, что также является полезным навыком. В современном мире часто возникает потребность в обработке больших объемов данных, так что умение работы с аллокаторами мне пригодится.

**Исходный код:**

**CMakeLists.txt:**

cmake\_minimum\_required(VERSION 3.20)

set(CMAKE\_CXX\_STANDARD 14)

add\_executable(lab6 main.cpp hexagon.cpp)

**figure.h:**

#ifndef FIGURE\_H

#define FIGURE\_H

class Figure {

public:

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

virtual size\_t VerticesNumber() = 0;

virtual double Area() = 0;

virtual ~Figure() {}

};

#endif

**hexagon.h:**

#ifndef HEXAGON\_H

#define HEXAGON\_H

#include <iostream>

#include "figure.h"

class Hexagon : public Figure{

public:

friend std::istream &operator >> (std::istream &is, Hexagon &p);

friend std::ostream &operator << (std::ostream &os, Hexagon &p);

Hexagon &operator = (const Hexagon &right);

bool operator == (const Hexagon &right);

size\_t VerticesNumber();

double Area();

private:

std::pair <double, double> p[6];

};

#endif

**tnarytree.h**:

#ifndef TNARYTREE\_H

#define TNARYTREE\_H

#include "hexagon.h"

#include<iostream>

#include <stdexcept>

#include <iterator>

#include <cstddef>

#include"tqueue.h"

using namespace std;

template <typename T>

struct myallocator {

const int num = 2;

TQueue <T \*> Free, Occupied;

void add\_space(){

T \* t = (T \*) malloc (sizeof(T) \* num);

for (int i = 0; i < num; i++) Free.push(t + i);

}

T \* get(){

// cout << "GET:" << endl;

if (Free.empty()) add\_space();

T \* res = Free.pop();

Occupied.push(res);

// cout << "FREE:" << endl;

// Free.print();

// cout << "BUSY:" << endl;

// Occupied.print();

// cout << "END" << endl;

return res;

}

void make\_free(T \* now){

// cout << "MAKE\_FREE: " << now << endl;

Occupied.find\_delete(now);

Free.push(now);

// cout << "FREE:" << endl;

// Free.print();

// cout << "BUSY:" << endl;

// Occupied.print();

// cout << "END" << endl;

}

};

template <typename T>

class node{

public:

shared\_ptr <node<T>> brt, cld;

shared\_ptr <T> val;

static myallocator <node <T> > alloc;

node(){

cld = NULL;

brt = NULL;

val = NULL;

}

friend std::ostream& operator<<(std::ostream& os, const node<T> &now);

void \* operator new (size\_t trash){

node <T> \* res = alloc.get();

return res;

}

void operator delete(void \* t){

alloc.make\_free((node <T> \*) t);

}

};

template <typename T>

myallocator <node <T> > node<T>::alloc;

template <typename T>

std::ostream& operator<<(std::ostream& os, const node<T> &now){

os << now->val << endl;

return os;

}

template <typename T>

class TNaryTree {

public:

struct Iterator {

using iterator\_category = std::forward\_iterator\_tag;

using difference\_type = std::ptrdiff\_t;

using value\_type = T;

using pointer = T\*;

using reference = T&;

Iterator(node<T> \* ptr) { m\_ptr = ptr; }

reference operator\*() const { return \*(m\_ptr->val.get()); }

pointer operator->() { return m\_ptr->val.get(); }

Iterator& operator++() {

if (q.empty() && m\_ptr->cld.get() == NULL && m\_ptr->brt.get() == NULL){

m\_ptr = NULL;

return \* this;

}

if (m\_ptr->cld.get() != NULL) q.push(m\_ptr->cld.get());

if (m\_ptr->brt.get() != NULL) q.push(m\_ptr->brt.get());

m\_ptr = q.pop();

return \* this;

}

Iterator operator++(int) { Iterator tmp = \*this; ++(\*this); return tmp; }

friend bool operator== (const Iterator& a, const Iterator& b) { return a.m\_ptr == b.m\_ptr; };

friend bool operator!= (const Iterator& a, const Iterator& b) { return a.m\_ptr != b.m\_ptr; };

private:

node<T> \* m\_ptr;

TQueue <node<T> \*> q;

};

Iterator begin() {

if (!this->Empty()) return Iterator(this->root.get());

return Iterator(NULL);

}

Iterator end() { return Iterator(NULL); }

// Инициализация дерева с указанием размера

TNaryTree(int);

// Полное копирование дерева

TNaryTree(const TNaryTree& 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.

void Update(T &now, 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);

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

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

template <typename U>

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

~TNaryTree();

private:

shared\_ptr <node<T>> root;

shared\_ptr <node<T>> now;

int size;

int max\_size;

int get\_size();

void check();

void decrease();

void node\_delete(shared\_ptr <node<T>>, TNaryTree &, shared\_ptr <node<T>>);

void change\_val(T &);

};

template <typename T>

void TNaryTree<T>::check(){

if (size >= max\_size){

throw out\_of\_range("Nodes limit exceeded.");

exit(-1);

}

}

template <typename T>

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

max\_size = n;

root = NULL;

size = 0;

}

template <typename T>

void assign (shared\_ptr <node<T>> left, T & right){

if (left->val == NULL){

T \* ptr = new T;

\* ptr = right;

shared\_ptr <T> temp(ptr);

left->val.swap(temp);

return;

}

\* left->val = right;

}

template <typename T>

shared\_ptr <node<T>> create\_node(shared\_ptr <node<T>> other){

node<T> \* temp = new node<T>;

shared\_ptr <node<T>> ptr(temp);

ptr->val = other->val;

return ptr;

}

template <typename T>

void TNaryTree<T>::decrease(){

size--;

}

template <typename T>

int TNaryTree<T>::get\_size(){

return size;

}

template <typename T>

void node\_copy(shared\_ptr <node<T>> now, const shared\_ptr <node<T>> other, char p){

if (other == NULL) return;

if (p == 'c') {

now->cld = create\_node(other);

now = now->cld;

}

else{

now->brt = create\_node(other);

now = now->brt;

}

node\_copy (now, other->cld, 'c');

node\_copy (now, other->brt, 'b');

}

// Полное копирование дерева

template <typename T>

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

if (other.root == NULL) return;

size = other.size;

max\_size = other.max\_size;

root = create\_node(other.root);

node\_copy (root, other.root->cld, 'c');

}

// 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.

template <typename T>

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

if (tree\_path == ""){

if (root == NULL){

check();

node<T> \* ptr = new node<T>;

shared\_ptr <node<T>> temp(ptr);

assign(temp, h);

root = temp;

size++;

}

else assign(root, h);

return;

}

if (root == NULL || tree\_path[0] == 'b'){

throw invalid\_argument("Invalid tree\_path.");

exit(-1);

}

now = root;

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

if (tree\_path[i] == 'c') now = now->cld;

else now = now->brt;

if (now == NULL){

throw invalid\_argument("Invalid tree\_path.");

exit(-1);

}

}

if (tree\_path[tree\_path.size() - 1] == 'c'){

if (now->cld == NULL){

check();

node<T> \* ptr = new node<T>;

shared\_ptr <node<T>> temp(ptr);

assign(temp, h);

now->cld = temp;

size++;

}

else assign(now->cld, h);

}

else{

if (now->brt == NULL){

check();

node<T> \* ptr = new node<T>;

shared\_ptr <node<T>> temp(ptr);

assign(temp, h);

now->brt = temp;

size++;

}

else assign(now->brt, h);

}

}

template <typename T>

void TNaryTree<T>::node\_delete(shared\_ptr <node<T>> now1, TNaryTree<T> &t, shared\_ptr <node<T>> rt){

if (now1==NULL) return;

t.decrease();

node\_delete(now1->cld, t, rt);

if (now1 != rt) node\_delete(now1->brt, t, rt);

}

// Удаление поддерева

template <typename T>

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

now = NULL;

if (tree\_path == ""){

node\_delete(root, \* this, root);

root = NULL;

return;

}

if (root == NULL || tree\_path[0] == 'b'){

throw invalid\_argument("Invalid tree\_path.");

exit(-1);

}

now = root;

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

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

now = now->cld;

}

else now = now->brt;

if (now == NULL){

throw invalid\_argument("Invalid tree\_path.");

exit(-1);

}

}

if (tree\_path[tree\_path.size() - 1] == 'c'){

if (now->cld == NULL){

throw invalid\_argument("Invalid tree\_path.");

exit(-1);

}

node\_delete (now->cld, \* this, now->cld);

now->cld = now->cld->brt;

}

else{

if (now->brt == NULL){

throw invalid\_argument("Invalid tree\_path.");

exit(-1);

}

node\_delete (now->brt, \* this, now->brt);

now->brt = now->brt->brt;

}

}

// Проверка наличия в дереве вершин

template <typename T>

bool TNaryTree<T>::Empty(){

return root == NULL;

}

template <typename T>

double node\_square(shared\_ptr <node<T>> now){

if (now == NULL) return 0;

double res = now->val->Area();

res += node\_square(now->cld);

res += node\_square(now->brt);

return res;

}

// Подсчет суммарной площади поддерева

template <typename T>

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

if (tree\_path == "") return node\_square(root);

if (root == NULL || tree\_path[0] == 'b'){

throw invalid\_argument("Invalid tree\_path.");

exit(-1);

}

now = root;

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

if (tree\_path[i] == 'c') now = now->cld;

else now = now->brt;

if (now == NULL){

throw invalid\_argument("Invalid tree\_path.");

exit(-1);

}

}

if (tree\_path[tree\_path.size() - 1] == 'c'){

if (now->cld == NULL){

throw invalid\_argument("Invalid tree\_path.");

exit(-1);

}

return node\_square (now->cld);

}

else{

if (now->brt == NULL){

throw invalid\_argument("Invalid tree\_path.");

exit(-1);

}

return node\_square (now->brt);

}

return 0;

}

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

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

template <typename T>

void print(std::ostream& os, shared\_ptr <node<T>> v, bool p){

os << v->val->Area();

if (v->cld != NULL){

os << ": [";

print(os, v->cld, 1);

}

if (v->brt != NULL){

os << ", ";

print(os, v->brt, 1);

}

else if (p) os << "]";

}

template <typename T>

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

if (tree.root == NULL) return os;

print(os, tree.root, 0);

return os;

}

template <typename T>

TNaryTree<T>::~TNaryTree(){

string s = "";

Clear(s);

}

#endif

**tqueue.h**:

#ifndef QUEUE\_H

#define QUEUE\_H

#include<iostream>

using namespace std;

template <typename T>

struct q\_node{

shared\_ptr <T> val;

shared\_ptr< q\_node<T> > next;

q\_node(T &now) {

T \* temp = new T;

\* temp = now;

val.reset(temp);

next = NULL;

}

~q\_node(){

val = NULL;

next = NULL;

// cout << "DELETED" << endl;

}

};

template <typename T>

struct TQueue{

public:

TQueue(){ head = NULL; }

void print(){

if (this->empty()){

cout << endl;

return;

}

q\_node<T> \* temp = this->head.get();

while (temp != NULL){

cout << \* temp->val << " ";

temp = temp->next.get();

}

cout << endl;

}

void push(T add){

if (this->empty()){

q\_node <T> \* temp = new q\_node<T>(add);

this->head.reset(temp);

}

else{

q\_node<T> \* now = this->head.get();

while (now->next != NULL) now = now->next.get();

q\_node <T> \* temp = new q\_node<T>(add);

now->next.reset(temp);

}

}

T pop(){

if (this->empty()) exit(-1);

T res = \* this->head->val.get();

shared\_ptr< q\_node<T> > temp(this->head->next);

this->head = temp;

temp = NULL;

return res;

}

bool empty(){ return head == NULL;}

void find\_delete(T t){

if (\* this->head.get()->val == t){

this->head = this->head->next;

return;

}

q\_node <T> \* now = this->head.get();

while (\* now->next->val != t){

if (now->next == NULL) exit(-1);

now = now->next.get();

}

now->next = now->next->next;

}

~TQueue(){

this->head = NULL;

}

private:

shared\_ptr <q\_node <T> > head;

};

#endif

**hexagon.cpp:**

#include"hexagon.h"

#include<iostream>

#include<math.h>

using namespace std;

std::istream &operator >> (std::istream &is, Hexagon &h){

for (int i = 0; i < 6; i++) is >> h.p[i].first >> h.p[i].second;

return is;

}

std::ostream &operator << (std::ostream &os, Hexagon &h){

os << "Hexagon:";

for (int i = 0; i < 6; i++) os << " (" << h.p[i].first << ", " << h.p[i].second << ")";

return os;

}

double hex\_distance(pair <double, double> a, pair<double, double> b){

return sqrt(pow(a.first - b.first, 2) + pow(a.second - b.second, 2));

}

double hex\_S\_triangle (double a, double b, double c){

double p = (a + b + c) / 2;

return sqrt(p \* (p - a) \* (p - b) \* (p - c));

}

size\_t Hexagon::VerticesNumber(){

return 6;

}

double Hexagon::Area(){

double res = 0, a, b, c;

for (int i = 1; i < 5; i++){

a = hex\_distance(this->p[0], this->p[i]);

b = hex\_distance(this->p[i], this->p[i + 1]);

c = hex\_distance(this->p[0], this->p[i + 1]);

res += hex\_S\_triangle(a, b, c);

}

return res;

}

Hexagon & Hexagon::operator = (const Hexagon &right){

if (this == &right) return \*this;

for (int i = 0; i < 6; i++) this->p[i] = right.p[i];

return \*this;

}

double find\_eps(){

double eps = 1;

while (eps + 1 != 1){

eps /= 2;

}

return eps;

}

double eps = find\_eps();

bool Hexagon::operator == (const Hexagon &right){

bool p = 1;

for (int i = 0; i < 6; i++){

if (abs(this->p[i].first - right.p[i].first) >= eps || abs(this->p[i].second - right.p[i].second) >= eps)

p = 0;

}

return p;

}

**main.cpp**:

#include<iostream>

#include"hexagon.h"

#include"tnarytree.h"

using namespace std;

int main(){

#include<iostream>

#include"hexagon.h"

#include"tnarytree.h"

using namespace std;

int main(){

cout << endl << "USAGE:" << endl;

cout << "Add element:" << endl;

cout << "a [x1] [y1] [x2] [y2] [x3] [y3] [x4] [y4] [x5] [y5] [x6] [y6]" << endl << "[route]" << endl;

cout << "example:" << endl << "a 0 2 1 1 1 -1 0 -2 -1 -1 -1 1" << endl << "cbb" << endl;

cout << "Print: p" << endl;

cout << "Delete:" << endl << "d" << endl << "[path]" << endl;

cout << "Area:" << endl << "s" << endl << "[path]" << endl;

cout << "Traversal with iterator: i" << endl;

cout << "Stop: q" << endl;

cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << endl;

int n;

cout << "Enter tree size:" << endl;

cin >> n;

TNaryTree<Hexagon> \* temp = new TNaryTree<Hexagon>(n);

shared\_ptr <TNaryTree<Hexagon>> t(temp);

char ch;

string s;

Hexagon h;

while (1){

cin >> ch;

switch (ch){

case 'a': {

cin >> h;

getline(cin, s);

getline(cin, s);

t->Update(h, s);

cout << "added" << endl;

break;

}

case 'p': {

cout << \* t << endl;

break;

}

case 'd': {

getline(cin, s);

getline(cin, s);

t->Clear(s);

cout << "deleted " << endl;

break;

}

case 's': {

getline(cin, s);

getline(cin, s);

cout << t->Area(s) << endl;

break;

}

case 'q': {

s = "";

t->Clear(s);

return 0;

}

case 'i':{

for (auto i : (\* temp)){

Hexagon qw = i;

cout << qw << endl;

}

}

}

}

return 0;

}