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: *Степанов Данила Михайлович, группа М8О-207Б-20*

Преподаватель: *Дорохов Евгений Павлович, каф. 806*

**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.

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.

Нельзя использовать:

* Стандартные контейнеры std.

Программа должна позволять:

* Вводить произвольное количество фигур и добавлять их в контейнер;
* Распечатывать содержимое контейнера;
* Удалять фигуры из контейнера.

**Вариант №24:**

* + Фигура: 8-угольник (Octagon)
  + Контейнер: N-дерево (TNaryTree)

**Описание программы:**

Исходный код разделён на 10 файлов:

* figure.h – описание класса фигуры
* point.h – описание класса точки
* point.cpp – реализация класса точки
* octagon.h – описание класса 8-угольника
* octagon.cpp – реализация класса 8-угольника
* TNaryTree\_item.h – описание элемента N-дерева
* TNaryTree.h – описание N-дерева
* TNaryTree.cpp – реализация N-дерева
* titerator.h – описание итератора
* main.cpp – основная программа
* tallocation\_block.h – описание аллокатора
* tallocation\_block.cpp – реализация аллокатора
* tstack\_item.h – описание элемента стека
* tstack\_item.cpp – реализация элемента стека
* tstack.h – описание стека
* tstack.cpp – реализация стека

**Дневник отладки:**

Проблем не возникло.

**Тестирование программы:**

The tree is empty !

0.5: [36: [12: [18, 19.5], 6.5], 7.5: [6, 16.5], 3.5: [21]]

44

0.5: [36: [12: [18, 19.5], 6.5], 7.5: [6, 16.5], 3.5: [21]]

Octagon: (1, 4) (1, 2) (5, 6) (2, 8) (3, 1) (2, 6) (9, 5) (5, 4)

The tree is not empty !

Allocation test:

Memory init

Allocate 1

Allocate 2

Allocate 3

10 100 1000

Free blocks are avaible !

**Вывод:**  
 В данной лабораторной я реализовал аллокатор памяти для динамических структур данных. Целью его построения является минимизации вызова операции malloc. Данная работа позволила мне закрепить навыки работы с памятью.

**Исходный код:**

**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 getX();

double getY();

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

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

private:

double x\_;

double y\_;

};

#endif

**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::getX()

{

return x\_;

}

double Point::getY()

{

return y\_;

}

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;

}

**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;

};

#endif

**octagon.h:**

#ifndef OCTAGON\_H

#define OCTAGON\_H

#include "point.h"

#include "figure.h"

#include "tallocation\_block.h"

class octagon : figure

{

public:

octagon(std::istream& is);

octagon();

~octagon();

octagon(Point a, Point b, Point c, Point d, Point e, Point f, Point g, Point h);

size\_t VertexesNumber();

double Area();

void Print(std::ostream& os);

octagon& operator=(const octagon& other);

bool operator==(octagon& other);

friend std::ostream& operator<<(std::ostream& os, octagon& other);

friend std::istream& operator>>(std::istream& is, octagon& other);

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

void operator delete(void\* ptr);

private:

Point a\_, b\_, c\_, d\_;

Point e\_, f\_, g\_, h\_;

static TAllocationBlock octagonallocator;

};

#endif

**octagon.cpp:**

#include "octagon.h"

#include "point.h"

octagon::octagon(std::istream& is)

{

std::cin >> a\_ >> b\_ >> c\_ >> d\_;

std::cin >> e\_ >> f\_ >> g\_ >> h\_;

}

octagon::octagon() : a\_(0,0), b\_(0,0), c\_(0,0), d\_(0, 0), e\_(0,0), f\_(0,0), h\_(0,0), g\_(0, 0)

{}

octagon::octagon(Point a, Point b, Point c, Point d, Point e, Point f, Point g, Point h)

{

this->a\_ = a; this->b\_ = b;

this->c\_ = c; this->d\_ = d;

this->e\_ = e; this->f\_ = f;

this->h\_ = h; this->g\_ = g;

}

size\_t octagon::VertexesNumber()

{

return (size\_t)8;

}

double octagon::Area()

{

return 0.5 \* abs((a\_.getX() \* b\_.getY() + b\_.getX() \* c\_.getY() + c\_.getX() \* d\_.getY() + d\_.getX() \* e\_.getY() + e\_.getX() \* f\_.getY() +

f\_.getX() \* g\_.getY() + g\_.getX() \* h\_.getY() + h\_.getX() \* a\_.getY() - (b\_.getX() \* a\_.getY() + c\_.getX() \* b\_.getY() +

d\_.getX() \* c\_.getY() + e\_.getX() \* d\_.getY() + f\_.getX() \* e\_.getY() + g\_.getX() \* f\_.getY() + h\_.getX() \* g\_.getY() +

a\_.getX() \* h\_.getY())));

}

octagon& octagon::operator=(const octagon& other)

{

this->a\_ = other.a\_; this->b\_ = other.b\_;

this->c\_ = other.c\_; this->d\_ = other.d\_;

this->e\_ = other.e\_; this->f\_ = other.f\_;

this->g\_ = other.g\_; this->h\_ = other.h\_;

return \*this;

}

bool octagon::operator==(octagon& other)

{

return this->a\_ == other.a\_ && this->b\_ == other.b\_ &&

this->c\_ == other.c\_ && this->d\_ == other.d\_ &&

this->e\_ == other.e\_ && this->f\_ == other.f\_ &&

this->g\_ == other.g\_ && this->h\_ == other.h\_;

}

std::ostream& operator<<(std::ostream& os, octagon& oct)

{

os << "Octagon: " << oct.a\_ << " " << oct.b\_ << " ";

os << oct.c\_ << " " << oct.d\_ << " " << oct.e\_ << " ";

os << oct.f\_ << " " << oct.g\_ << " " << oct.h\_ << "\n";

return os;

}

std::istream& operator>>(std::istream& is, octagon& other)

{

is >> other.a\_ >> other.b\_ >> other.c\_ >> other.d\_;

is >> other.e\_ >> other.f\_ >> other.g\_ >> other.h\_;

return is;

}

void octagon::Print(std::ostream& os)

{

std::cout << "Octagon: " << a\_ << " " << b\_ << " ";

std::cout << c\_ << " " << d\_ << " " << e\_ << " ";

std::cout << f\_ << " " << g\_ << " " << h\_ << "\n";

}

TAllocationBlock octagon::octagonallocator(sizeof(octagon), 10);

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

return octagonallocator.allocate();

}

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

octagonallocator.deallocate(ptr);

}

octagon::~octagon(){}

**TNaryTree\_item.h:**

#ifndef TNARYTREE\_ITEM

#define TNARYTREE\_ITEM

#include "octagon.h"

#include <memory>

template<class T>

class TreeItem

{

public:

std::shared\_ptr<T> figure;

int cur\_size;

std::shared\_ptr<TreeItem<T>> son;

std::shared\_ptr<TreeItem<T>> brother;

std::shared\_ptr<TreeItem<T>> parent;

};

#endif

**TnaryTree.h:**

#ifndef TNARY\_TREE

#define TNARY\_TREE

#include "octagon.h"

#include "TNaryTree\_item.h"

#include <memory>

template<class T>

class TNaryTree

{

public:

TNaryTree(int n);

TNaryTree(const TNaryTree<T>& other);

TNaryTree();

void Update(const std::shared\_ptr<T> &&polygon, const std::string &&tree\_path)

{

Update(&root, polygon, tree\_path);

}

void Update(const std::shared\_ptr<T> &polygon, const std::string &tree\_path)

{

Update(&root, polygon, tree\_path);

}

const std::shared\_ptr<T>& GetItem(const std::string& tree\_path)

{

return GetItem(&root, tree\_path);

}

void RemoveSubTree(const std::string &&tree\_path);

void RemoveSubTree(const std::string &tree\_path);

bool Empty();

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

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

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

virtual ~TNaryTree();

private:

int size;

std::shared\_ptr<TreeItem<T>> root;

void Update(std::shared\_ptr<TreeItem<T>>\* root, std::shared\_ptr<T> polygon, std::string tree\_path);

const std::shared\_ptr<T>& GetItem(std::shared\_ptr<TreeItem<T>>\* root, const std::string tree\_path);

};

#endif

**TNaryTree.cpp:**

#include "TNaryTree.h"

#include "TNaryTree\_item.h"

template<class T>

TNaryTree<T>::TNaryTree(int n)

{

this->size = n;

this->root = nullptr;

}

template<class T>

std::shared\_ptr<TreeItem<T>> tree\_copy(std::shared\_ptr<TreeItem<T>> root)

{

if (root != nullptr) {

std::shared\_ptr<TreeItem<T>> new\_root (new TreeItem<T>);

new\_root->figure = root->figure;

new\_root->son = nullptr;

new\_root->brother = nullptr;

if (root->son != nullptr) {

new\_root->son = tree\_copy(root->son);

}

if (root->brother != nullptr) {

new\_root->brother = tree\_copy(root->brother);

}

return new\_root;

}

return nullptr;

}

template<class T>

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

{

this->root = tree\_copy(other.root);

this->root->cur\_size = 0;

this->size = other.size;

}

template<class T>

void TNaryTree<T>::Update(std::shared\_ptr<TreeItem<T>>\* root, std::shared\_ptr<T> polygon, std::string tree\_path)

{

if (tree\_path == "") {

if (\*root == nullptr) {

\*root = std::shared\_ptr<TreeItem<T>>(new TreeItem<T>);

(\*root)->figure = std::shared\_ptr<T>(new T);

(\*root)->figure = polygon;

(\*root)->brother = nullptr;

(\*root)->son = nullptr;

(\*root)->parent = nullptr;

} else {

(\*root)->figure = polygon;

}

return;

}

if (tree\_path == "b") {

std::cout << "Cant add brother to root\n";

return;

}

std::shared\_ptr<TreeItem<T>> cur = \*root;

if (cur == NULL) {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

return;

}

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

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

cur = cur->son;

} else {

cur = cur->brother;

}

if (cur == nullptr && i < tree\_path.size() - 1) {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

return;

}

}

if (tree\_path[tree\_path.size() - 1] == 'c' && cur->son == nullptr) {

if (cur->cur\_size + 1 > this->size) {

throw std::out\_of\_range("Tree is overflow\n");

return;

}

if (cur->son == nullptr) {

cur->son = std::shared\_ptr<TreeItem<T>>(new TreeItem<T>);

cur->son->figure = std::shared\_ptr<T>(new T);

cur->son->figure = polygon;

cur->son->son = nullptr;

cur->son->brother = nullptr;

cur->son->parent = cur;

cur->son->parent->cur\_size++;

} else {

cur->son->figure = polygon;

}

} else if (tree\_path[tree\_path.size() - 1] == 'b' && cur->brother == nullptr) {

if (cur->parent->cur\_size + 1 > this->size) {

throw std::out\_of\_range("Tree is overflow\n");

return;

}

if (cur->brother == nullptr) {

cur->brother = std::shared\_ptr<TreeItem<T>>(new TreeItem<T>);

cur->brother->figure = std::shared\_ptr<T>(new T);

cur->brother->figure = polygon;

cur->brother->son = nullptr;

cur->brother->brother = nullptr;

cur->brother->parent = cur->parent;

cur->brother->parent->cur\_size++;

} else {

cur->brother->figure = polygon;

}

}

}

template<class T>

void delete\_tree(std::shared\_ptr<TreeItem<T>>\* root)

{

if ((\*root)->son != nullptr) {

delete\_tree(&((\*root)->son));

}

if ((\*root)->brother != nullptr) {

delete\_tree(&((\*root)->brother));

}

\*root = nullptr;

}

template<class T>

void delete\_undertree(std::shared\_ptr<TreeItem<T>>\* root, char c)

{

if (\*root == nullptr) {

return;

}

if (c == 'b') {

if ((\*root)->brother != nullptr) {

std::shared\_ptr<TreeItem<T>> cur = (\*root)->brother;

if ((\*root)->brother->brother != nullptr) {

(\*root)->brother = (\*root)->brother->brother;

cur->brother = nullptr;

delete\_tree(&cur);

} else {

delete\_tree(&((\*root)->brother));

}

}

} else if (c == 'c') {

std::shared\_ptr<TreeItem<T>> cur = (\*root)->son;

if ((\*root)->son->brother != nullptr) {

(\*root)->son = (\*root)->son->brother;

if (cur->son != nullptr) {

delete\_tree(&(cur->son));

}

cur = nullptr;

} else {

delete\_tree(&((\*root)->son));

}

}

}

template<class T>

void TNaryTree<T>::RemoveSubTree(const std::string &&tree\_path)

{

if (tree\_path == "" && this->root != nullptr) {

std::shared\_ptr<TreeItem<T>>\* iter = &(this->root);

delete\_tree(iter);

return;

} else if (tree\_path == "" && this->root == nullptr) {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

return;

}

std::shared\_ptr<TreeItem<T>> cur = this->root;

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

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

if (cur->son == nullptr) {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

return;

}

cur = cur->son;

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

if (cur->brother == nullptr) {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

return;

}

cur = cur->brother;

}

}

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

if (cur->son == nullptr) {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

return;

}

delete\_undertree(&cur, 'c');

} else if (tree\_path[tree\_path.size() - 1] == 'b') {

if (cur->brother == nullptr) {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

return;

}

delete\_undertree(&cur, 'b');

}

return;

}

template<class T>

void TNaryTree<T>::RemoveSubTree(const std::string &tree\_path)

{

if (tree\_path == "" && this->root != nullptr) {

std::shared\_ptr<TreeItem<T>>\* iter = &(this->root);

delete\_tree(iter);

return;

} else if (tree\_path == "" && this->root == nullptr) {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

return;

}

std::shared\_ptr<TreeItem<T>> cur = this->root;

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

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

if (cur->son == nullptr) {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

return;

}

cur = cur->son;

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

if (cur->brother == nullptr) {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

return;

}

cur = cur->brother;

}

}

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

if (cur->son == nullptr) {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

return;

}

delete\_undertree(&cur, 'c');

} else if (tree\_path[tree\_path.size() - 1] == 'b') {

if (cur->brother == nullptr) {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

return;

}

delete\_undertree(&cur, 'b');

}

return;

}

template<class T>

bool TNaryTree<T>::Empty()

{

if (this->root != nullptr) {

return false;

} else {

return true;

}

}

template<class T>

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

{

if (tree\_path == "") {

if (this->root != nullptr) {

return this->root->figure->Area();

} else {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

}

}

std::shared\_ptr<TreeItem<T>> cur = this->root;

double square = 0;

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

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

if (cur->son != nullptr) {

cur = cur->son;

} else {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

}

} else {

if (cur->brother != nullptr) {

cur = cur->brother;

} else {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

}

}

square += cur->figure->Area();

}

return square + this->root->figure->Area();

}

template<class T>

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

{

if (tree\_path == "") {

if (this->root != nullptr) {

return this->root->figure->Area();

} else {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

}

}

std::shared\_ptr<TreeItem<T>> cur = this->root;

double square = 0;

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

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

if (cur->son != nullptr) {

cur = cur->son;

} else {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

}

} else {

if (cur->brother != nullptr) {

cur = cur->brother;

} else {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

}

}

square += cur->figure->Area();

}

return square + this->root->figure->Area();

}

template<class T>

void Print(std::ostream& os, std::shared\_ptr<TreeItem<T>> vertex)

{

if (vertex != nullptr) {

os << vertex->figure->Area();

if (vertex->son != nullptr) {

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

Print(os, vertex->son);

if ((vertex->son->brother == nullptr && vertex->brother != nullptr) || (vertex->son->brother == nullptr && vertex->brother == nullptr)) {

os << "]";

}

}

if (vertex->brother != nullptr) {

os << ", ";

Print(os, vertex->brother);

if (vertex->brother->brother == nullptr) {

os << "]";

}

}

} else {

return;

}

}

template<class A>

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

{

if (tree.root != nullptr) {

Print(os, tree.root); os << "\n";

return os;

} else {

os << "Tree has no vertex\n";

return os;

}

}

template<class T>

const std::shared\_ptr<T>& TNaryTree<T>::GetItem(std::shared\_ptr<TreeItem<T>>\* root, const std::string tree\_path)

{

if (tree\_path == "" && \*root == nullptr) {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

}

std::shared\_ptr<TreeItem<T>> cur = \*root;

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

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

if (cur->son == nullptr) {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

}

cur = cur->son;

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

if (cur->brother == nullptr) {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

}

cur = cur->brother;

}

}

return cur->figure;

}

template<class T>

TNaryTree<T>::~TNaryTree()

{

if (this->root != nullptr) {

this->RemoveSubTree("");

}

}

template class TNaryTree<octagon>;

template std::ostream& operator<< <octagon>(std::ostream&, TNaryTree<octagon> const&);

**titerator.h:**

#ifndef TITERATOR\_H

#define TITERATOR\_H

#include <iostream>

#include <memory>

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->figure;}

std::shared\_ptr<T> operator->() {return node\_ptr->figure;}

void operator++() {node\_ptr = node\_ptr->GetNext();}

TIterator operator++(int) {

TIterator iter(\*this);

++(\*this);

return iter;

}

bool operator==(TIterator const& i) { return node\_ptr == i.node\_ptr; }

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

private:

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

};

#endif

**tallocation\_block.h:**

#ifndef TALLOCATION\_BLOCK\_H

#define TALLOCATION\_BLOCK\_H

#include <cstdlib>

#include "tstack.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;

TStack<void\*> \_free\_blocks;

size\_t \_free\_count;

};

#endif

**tallocation\_block.cpp:**

#include "tallocation\_block.h"

#include <iostream>

TAllocationBlock::TAllocationBlock(size\_t size, size\_t count): \_size(size), \_count(count)

{

\_used\_blocks = (char\*)malloc(size \* count);

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

\_free\_blocks.Push(\_used\_blocks + i \* size);

}

\_free\_count = count;

std::cout << "Memory init" << "\n";

}

void\* TAllocationBlock::allocate()

{

void\* result = nullptr;

if (\_free\_count == 0) {

std::cout << "No memory exception\n" << "\n";

return result;

}

result = \_free\_blocks.Top();

\_free\_blocks.Pop();

--\_free\_count;

std::cout << "Allocate " << (\_count - \_free\_count) << "\n";

return result;

}

void TAllocationBlock::deallocate(void\* pointer)

{

\_free\_blocks.Push(pointer);

++\_free\_count;

std::cout << "Deallocated block\n";

}

bool TAllocationBlock::has\_free\_blocks()

{

return \_free\_count > 0;

}

TAllocationBlock::~TAllocationBlock()

{

free(\_used\_blocks);

}

**tstack\_item.h:**

#ifndef TSTACKITEM\_H

#define TSTACKITEM\_H

#include <iostream>

#include <memory>

template <class T>

class TStackItem

{

public:

TStackItem(const T &val, TStackItem<T> \*item);

virtual ~TStackItem();

void Push(const T &val);

T &Pop() const;

void SetNext(TStackItem<T> \*item);

TStackItem<T> &GetNext() const;

private:

T \*value;

TStackItem<T> \*next;

};

#endif

**tstack\_item.cpp:**

#include <iostream>

#include <memory>

#include "tstack\_item.h"

template <class T>

TStackItem<T>::TStackItem(const T &val, TStackItem<T> \*item)

{

value = new T(val);

next = item;

}

template <class T>

void TStackItem<T>::Push(const T &val)

{

\*value = val;

}

template <class T>

T &TStackItem<T>::Pop() const

{

return \*value;

}

template <class T>

void TStackItem<T>::SetNext(TStackItem<T> \*item)

{

next = item;

}

template <class T>

TStackItem<T> &TStackItem<T>::GetNext() const

{

return \*next;

}

template <class T>

TStackItem<T>::~TStackItem()

{

delete value;

}

template class

TStackItem<void \*>;

**tstack.h:**

#ifndef TSTACK\_H

#define TSTACK\_H

#include <iostream>

#include <memory>

#include "tstack\_item.h"

template <class T>

class TStack

{

public:

TStack();

virtual ~TStack();

void Push(const T &item);

void Pop();

T &Top();

bool IsEmpty() const;

uint32\_t GetSize() const;

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

private:

TStackItem<T> \*head;

uint32\_t count;

};

#endif

**tstack.cpp:**

#include <iostream>

#include <memory>

#include "tstack.h"

template <class T>

TStack<T>::TStack()

{

head = nullptr;

count = 0;

}

template <class T>

void TStack<T>::Push(const T &item)

{

TStackItem<T> \*tmp = new TStackItem<T>(item, head);

head = tmp;

++count;

}

template <class T>

bool TStack<T>::IsEmpty() const

{

return !count;

}

template <class T>

uint32\_t TStack<T>::GetSize() const

{

return count;

}

template <class T>

void TStack<T>::Pop()

{

if(head) {

TStackItem<T> \*tmp = &head->GetNext();

delete head;

head = tmp;

--count;

}

}

template <class T>

T &TStack<T>::Top()

{

return head->Pop();

}

template <class T>

TStack<T>::~TStack()

{

for(TStackItem<T> \*tmp = head, \*tmp2; tmp; tmp = tmp2) {

tmp2 = &tmp->GetNext();

delete tmp;

}

}

template class

TStack<void \*>;

**main.cpp:**

#include "TNaryTree.h"

#include "octagon.h"

#include "titerator.h"

#include "TNaryTree\_item.h"

#include "tallocation\_block.h"

#include <string>

int main()

{

TAllocationBlock block(sizeof(int), 10);

int\* n1;

int\* n2;

int\* n3;

n1 = (int\*)block.allocate();

n2 = (int\*)block.allocate();

n3 = (int\*)block.allocate();

octagon\* f1 = new octagon(Point(1, 1),Point(2, 2),Point(3, 3),Point(4, 4),

Point(5, 5),Point(6, 6),Point(7, 7),Point(8, 8));

octagon\* f2 = new octagon(Point(9, 9),Point(10, 10),Point(11, 11),Point(12, 12),

Point(13, 13),Point(14, 14),Point(15, 15),Point(16, 16));

octagon\* f3 = new octagon(Point(17, 17),Point(18, 18),Point(19, 19),Point(20, 20),

Point(21, 21),Point(22, 22),Point(23, 23),Point(24, 24));

(\*f1).Print(std::cout);

(\*f2).Print(std::cout);

(\*f3).Print(std::cout);

delete f1;

delete f2;

delete f3;

\*n1 = 10; \*n2 = 100; \*n3 = 1000;

std::cout << \*n1 << " " << \*n2 << " " << \*n3 << "\n";

if (block.has\_free\_blocks()) {

std::cout << "Free blocks are avaible !\n";

} else {

std::cout << "Free blocks are not avaible!\n";

}

return 0;

}