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;
}
```