МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ (НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСТИТЕТ)

ЛАБОРАТОРНАЯ РАБОТА №8

по курсу объектно-ориентированное программирование I семестр, 2021/22 уч. год

Студент: *Попов Матвей Романович, группа М8О-208Б-20*
Преподаватель: *Дорохов Евгений Павлович*

Задание

Используя структуру данных, разработанную для лабораторной работы №5, спроектировать и разработать аллокатор памяти для динамической структуры данных.

Вариант 18

Фигура треугольник, структура первого уровня бинарное дерево, структура второго уровня очередь.

Описание программы

Программа состоит из 15 файлов: main.cpp, figure.h, point.h, point.cpp, TBinaryTree.h, TBinaryTreeItem.h, TBinaryTree.cpp, TBinaryTree.cpp, triangle.h, triangle.cpp, TIterator.h, tqueue.hpp, tqueue_item.hpp, tallocation_block.h, tallocation_block.cpp, содержит аллокатор для динамической структуры данных.

Дневник отладки

При отладке ошибок в выполнении программы не выявлено.

Выводы

Проделав лабораторную работу, познакомился с аллокаторами в C++.

Листинг

main.cpp

```
#include "triangle.h" //g++ main.cpp point.cpp triangle.cpp tallocation_block.cpp -Wall -Wextra
-o main
#include <iostream>
#include <string>

int main()
{
    Point x1(0, 0);
    Point x2(1, 0);
    Point x3(0, 1);
    Triangle *t1 = new Triangle(x1, x2, x3);
    Triangle *t2 = new Triangle(x2, x1, x3);
    Triangle *t3 = new Triangle(x3, x1, x2);
    std::cout << "Three triangles have been initialized\n";
    delete t1;
    delete t2;
    delete t3;
    std::cout << "Three triangles have been deleted" << std::endl;
    return 0;
}
```

figure.h

```
#ifndef FIGURE_H
#define FIGURE_H

#include <cstddef>
#include "point.h"

using namespace std;

class Figure
{
public:
    virtual ~Figure()
    {};
    virtual double Area() = 0;
    virtual void Print(ostream& os) = 0;
    virtual size_t VertexesNumber() = 0;
};

#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);
}

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;
}
```

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);

  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_H
```

TBinaryTreeItem.cpp

```
#include "TBinaryTreeItem.h"


template <class T>
TBinaryTreeItem<T>::TBinaryTreeItem(const T &t)
{
    this->tri = t;
    this->left = NULL;
    this->right = NULL;
    this->counter = 1;
}

template <class T>
TBinaryTreeItem<T>::TBinaryTreeItem(const TBinaryTreeItem<T> &other)
{
    this->tri = other.tri;
    this->left = other.left;
    this->right = other.right;
    this->counter = other.counter;
}

template <class T>
TBinaryTreeItem<T>::~TBinaryTreeItem()
{}

template <class TT>
ostream& operator<<(ostream& os, TBinaryTreeItem<TT> tr)
{
    os << tr.tri << " ";
    return os;
}

#include "triangle.h"
template class TBinaryTreeItem<Triangle>;
template ostream& operator<<(ostream& os,  TBinaryTreeItem<Triangle> t);
```

TBinaryTreeItem.h

```
#ifndef TBINARYTREE_ITEM_H
#define TBINARYTREE_ITEM_H

#include "triangle.h"

template<class T>
class TBinaryTreeItem
{
public:
    TBinaryTreeItem(const T& tri);
    TBinaryTreeItem(const TBinaryTreeItem<T>& other);
```

```
    virtual ~TBinaryTreeItem();
    T tri;
    shared_ptr<TBinaryTreeItem<T>> left;
    shared_ptr<TBinaryTreeItem<T>> right;
    unsigned counter;

    template<class TT>
    friend ostream &operator<<(ostream &os, const TBinaryTreeItem<TT> &t);
};

#endif
```

TBinaryTree.h

```
#ifndef TBINARYTREE_H
#define TBINARYTREE_H

#include "TBinaryTreeItem.h"

using namespace std;

template <class T>
class TBinaryTree
{
private:
    shared_ptr <TBinaryTreeItem<T>> node;

public:
    TBinaryTree();
    void Push(const T& tr);
    const T& GetItemNotLess(double area);
    size_t Count(const T& t);
    void Pop(const T& t);
    bool Empty();
    template <class TT>
    friend ostream& operator<<(ostream& os, const TBinaryTree<TT>& tree);
    void Clear();
    virtual ~TBinaryTree();
};

#endif
```

TBinaryTree.cpp

```
#include "TBinaryTree.h"

using namespace std;

template <class T>
TBinaryTree<T>::TBinaryTree() : node(NULL)
{}

template <class T>
void print_tree(ostream& os, shared_ptr <TBinaryTreeItem<T>> node)
{
    if (!node)
    {
        return;
    }
    if (node->left)
    {
        os << node->counter << "*" << node->tri.GetArea() << ": [";
        print_tree(os, node->left);
```

```
        if (node->right)
        {
            os << ", ";
            print_tree(os, node->right);
        }
        os << "]";
    }
    else if (node->right)
    {
       os << node->counter << "*" << node->tri.GetArea() << ": [";
        print_tree(os, node->right);
        if (node->left)
        {
            os << ", ";
            print_tree(os, node->left);
        }
        os << "]";
    }
    else
    {
        os << node->counter << "*" << node->tri.GetArea();
    }
}

template <class TT>
std::ostream& operator << (ostream& os, const TBinaryTree<TT>& tree)
{
    print_tree(os, tree.node);
    os;
    return os;
}

template <class T>
void TBinaryTree<T>::Push(const T &tr)
{
    T t = tr;
    if (node == NULL)
    {
        shared_ptr <TBinaryTreeItem<T>> c(new TBinaryTreeItem<T>(t));
        node = c;
    }
    else if (node->tri.GetArea() == t.GetArea())
    {
        node->counter++;
    }
    else
    {
        shared_ptr<TBinaryTreeItem<T>> prev = node;
        shared_ptr<TBinaryTreeItem<T>> cur;
        bool bebra = true;
        if (t.GetArea() < prev->tri.GetArea())
        {
            cur = node->left;
        }
        else if (t.GetArea() > prev->tri.GetArea())
        {
            cur = node->right;
            bebra = false;
        }
        while (cur != NULL)
        {
            if (cur->tri == t)
            {
```

```
                cur->counter++;
            }
            else
            {
                if (t.GetArea() < cur->tri.GetArea())
                {
                    prev = cur;
                    cur = prev->left;
                    bebra = true;
                }
                else if (t.GetArea() > cur->tri.GetArea())
                {
                    prev = cur;
                    cur = prev->right;
                    bebra = false;
                }
            }
        }
        shared_ptr<TBinaryTreeItem<T>> c(new TBinaryTreeItem<T>(t));
        cur = c;
        if (bebra == true)
        {
            prev->left = cur;
        }
        else
        {
            prev->right = cur;
        }
    }
}

template <class T>
shared_ptr<TBinaryTreeItem<T>> __Pop(shared_ptr<TBinaryTreeItem<T>> node)
{
    if (node->left == NULL)
    {
        return node;
    }
    return __Pop(node->left);
}

template <class T>
shared_ptr<TBinaryTreeItem<T>> _Pop(shared_ptr<TBinaryTreeItem<T>> node, T &t)
{
    if (node == NULL)
    {
        return node;
    }
    else if (t.GetArea() < node->tri.GetArea())
    {
        node->left = _Pop(node->left, t);
    }
    else if (t.GetArea() > node->tri.GetArea())
    {
        node->right = _Pop(node->right, t);
    }
    else
    {
        if (node->left == NULL && node->right == NULL)
        {
            if (node->counter > 1)
            {
                --node->counter;
```

```
                return node;
            }
            node = NULL;
            return node;
        }
        else if (node->left == NULL && node->right != NULL)
        {
            if (node->counter > 1)
            {
                --node->counter;
                return node;
            }
            node = node->right;
            node->right = NULL;
            return node;
        }
        else if (node->right == NULL && node->left != NULL)
        {
            if (node->counter > 1)
            {
                --node->counter;
                return node;
            }
            node = node->left;
            node->left = NULL;
            return node;
        }
        else
        {
            shared_ptr<TBinaryTreeItem<T>> bebra = __Pop(node->right);
            node->tri.A = bebra->tri.GetArea();
            node->right = _Pop(node->right, bebra->tri);
        }
    }
    return node;
}

template <class T>
void TBinaryTree<T>::Pop(const T &t)
{
    T tr = t;
    node = _Pop(node, tr);
}

template <class T>
unsigned _Count(shared_ptr<TBinaryTreeItem<T>> cur, unsigned res, T& t)
{
    if (cur != NULL)
    {
        _Count(cur->left, res, t);
        _Count(cur->right, res, t);
        if (cur->tri.GetArea() == t.GetArea())
        {
            return cur->counter;
        }
    }
    return 0;
}

template <class T>
size_t TBinaryTree<T>::Count(const T& t)
{
    T tr = t;
```

```
    return _Count(node, 0, tr);
}

template <class T>
T& _GetItemNotLess(double area, shared_ptr<TBinaryTreeItem<T>> node)
{
    if (node->tri.GetArea() >= area)
    {
        return node->tri;
    }
    else
    {
        _GetItemNotLess(area, node->right);
    }
}

template <class T>
const T& TBinaryTree<T>::GetItemNotLess(double area)
{
    return _GetItemNotLess(area, node);
}

template <class T>
void _Clear(shared_ptr<TBinaryTreeItem<T>> cur)
{
    if (cur!= NULL)
    {
        _Clear(cur->left);
        _Clear(cur->right);
        cur = NULL;
    }
}

template <class T>
void TBinaryTree<T>::Clear()
{
    _Clear(node);
    node = NULL;
}

template <class T>
bool TBinaryTree<T>::Empty()
{
    return (node == NULL);
}

template <class T>
TBinaryTree<T>::~TBinaryTree()
{
    Clear();
}

template class TBinaryTree<Triangle>;
template ostream& operator<<(ostream& os, const TBinaryTree<Triangle>& tr);
```

triangle.h

```
#ifndef TRIANGLE_H
#define TRIANGLE_H

#include <iostream>
#include "figure.h"
```

```
using namespace std;

class Triangle : public Figure
{
private:
    Point p1, p2, p3;
public:
    Triangle();
    Triangle(istream& is);
    double Area();
    void Print(ostream& os);
    size_t VertexesNumber();
    virtual ~Triangle();
};

#endif
```

triangle.cpp

```
#include <cmath>
#include "triangle.h"

using namespace std;

Triangle::Triangle(istream& is)
{
    is >> p1 >> p2 >> p3;
}

void Triangle::Print(ostream& os)
{
    os << "Triangle: " << p1 << " " << p2 << " " << p3 << endl;
}

double Triangle::Area()
{
    double a = p1.dist(p2);
    double b = p2.dist(p3);
    double c = p3.dist(p1);
    double p = (a + b + c)/2;
    double s = sqrt(p * (p - a) * (p - b) * (p - c));
    return s;
}

size_t Triangle::VertexesNumber()
{
    return 3;
}

Triangle::~Triangle()
{
    cout << "Done\n";
}
```

TIterator.h

```
#ifndef TITERATOR_H
#define TITERATOR_H

#include <memory>
#include "TBinaryTreeItem.h"
#include "TBinaryTree.h"
```

```
template <class Node, class T>
class TIterator
{
private:
    std::shared_ptr<Node> node;

public:
    TIterator(std::shared_ptr<Node> n)
    {
        node = n;
    }

    T& operator*()
    {
        return node->tri;
    }

    void Left()
    {
        if (node == NULL)
        {
            return;
        }
        node = node->left;
    }

    void Right()
    {
        if (node == NULL)
        {
            return;
        }
        node = node->right;
    }

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

    bool operator!= (TIterator &i)
    {
        return !(node == i.node);
    }


};

#endif
```

tqueue.hpp

```
#ifndef TQueue_HPP
#define TQueue_HPP

#include "tqueue_item.hpp"

template <typename T>
class TQueue
{
public:
    TQueue()
    {
```

```
        heap = new TQueueItem<T>[max_length];
    }

    TQueue(const TQueue &o)
        :      heap(o.heap),      element_size(o.element_size),      max_length(o.max_length),
length(o.max_length) {}

    void Push(const T &item)
    {
        if (length >= max_length - 1)
        {
            max_length += 100;
            TQueueItem<T> *heap2 = new TQueueItem<T>[max_length];
            for (size_t i = 0; i < length; ++i)
            {
                heap2[i] = heap[i];
            }
            free(heap);
            heap = heap2;
        }
        TQueueItem<T> n(item);
        int input_pos, parent_pos;
        input_pos = length;
        heap[input_pos] = n;
        parent_pos = (input_pos - 1) / 2;
        while (parent_pos >= 0 && input_pos > 0)
        {
            TQueueItem<T> temp = heap[input_pos];
            heap[input_pos] = heap[parent_pos];
            heap[parent_pos] = temp;
            input_pos = parent_pos;
            parent_pos = (input_pos - 1) / 2;
        }
        ++length;
    }

    void Pop()
    {
        if (length == 0)
        {
            return;
        }
        heap[0] = heap[length - 1];
        --length;
        Heapify(0);
    }

    T Top() const
    {
        if (length == 0)
        {
            std::cout << "\nError: Queue is empty" << std::endl;
            exit(EXIT_FAILURE);
        }
        return heap[0].GetObject();
    }

    bool Empty() const
    {
        return length == 0;
    }

    size_t Length() const
```

```
    {
        return length;
    }

    template <typename A>
    friend std::ostream& operator<<(std::ostream &os, const TQueue<A> &_queue)
    {
        size_t i = 0, k = 1;
        while (i < _queue.length)
        {
            while ((i < k) && (i < _queue.length))
            {
                os << _queue.heap[i] << "\t";
                ++i;
            }
            if (i != _queue.length)
            {
                os << std::endl;
            }
            k = k * 2 + 1;
        }
        return os;
    }

    void Clear()
    {
        while (length > 0)
        {
            Pop();
        }
    }

    ~TQueue() {}

private:
    void Heapify(const int &position)
    {
        size_t left = 2 * position + 1, right = 2 * position + 2;
        if (left < length)
        {
            TQueueItem<T> tmp = heap[position];
            heap[position] = heap[left];
            heap[left] = tmp;
            Heapify(left);
        }
        if (right < length)
        {
            TQueueItem<T> tmp = heap[position];
            heap[position] = heap[right];
            heap[right] = tmp;
            Heapify(right);
        }
    }
    TQueueItem<T> *heap;
    const int element_size = sizeof(TQueueItem<T>);
    size_t max_length = 100;
    size_t length = 0;
};

#endif
```

tqueue_item.hpp

```
#ifndef TQueue_ITEM_HPP
#define TQueue_ITEM_HPP

#include <iostream>

template <typename T>
class TQueueItem
{
public:
    TQueueItem() = default;

    TQueueItem(const T &item) : item(item) {}

    TQueueItem(const TQueueItem<T> &other) : item(other.item) {}

    T GetObject() const
    {
        return item;
    }

    TQueueItem<T> &operator=(const TQueueItem<T> &other)
    {
        this->item = other.item;
        return *this;
    }

    bool operator==(const TQueueItem<T> &other) const
    {
        return (item == other.item);
    }

    bool operator!=(const TQueueItem<T> &other) const
    {
        return (item != other.item);
    }

    ~TQueueItem() {}

private:
    T item;
};

#endif
```

tallocation_block.h

```
#ifndef TALLOCATION_BLOCK_H
#define TALLOCATION_BLOCK_H

#include "tqueue.hpp"

class TAllocationBlock
{
public:
    TAllocationBlock(const size_t &size, const size_t &count);
    void* Allocate(const size_t &size_of_block);
    void Deallocate(void *pointer);
    bool HasFreeBlocks();
    virtual ~TAllocationBlock();

private:
    size_t size;
    size_t count;
```

```
    size_t free_count;
    char *used_blocks;
    TQueue<void*> q_free_blocks;
};

#endif
```

tallocation_block.cpp

```
#include "tallocation_block.h"

TAllocationBlock::TAllocationBlock(const size_t &size, const size_t &count) : size(size),
count(count)
{
    used_blocks = (char *)malloc(size * count);
    for (size_t i = 0; i < count; ++i)
    {
        q_free_blocks.Push(used_blocks + i * size);
    }
    free_count = count;
}

void* TAllocationBlock::Allocate(const size_t &size_of_block)
{
    if (size != size_of_block)
    {
        std::cout << "Error" << std::endl;
    }
    void *result = nullptr;
    if (free_count == 0)
    {
        size_t old_count = count;
        count += 10;
        free_count += 10;
        used_blocks = (char*) realloc(used_blocks, size * count);
        for (size_t i = old_count; i < count; ++i)
        {
            q_free_blocks.Push(used_blocks + i * size);
        }
    }
    result = q_free_blocks.Top();
    q_free_blocks.Pop();
    --free_count;
    return result;
}

void TAllocationBlock::Deallocate(void *pointer)
{
    q_free_blocks.Push(pointer);
    ++free_count;
}

bool TAllocationBlock::HasFreeBlocks()
{
    return free_count > 0;
}

TAllocationBlock::~TAllocationBlock()
{
    free(used_blocks);
}
```