МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСТИТЕТ)

ЛАБОРАТОРНАЯ РАБОТА №6

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

Студент *Шандрюк Пётр Николаевич, группа М8О-208Б-20*

Преподаватель *Дорохов Евгений Павлович*

Условие

Задание: Вариант 25: Труегольник, очередь. Используя структуру данных, разработанную для лабораторной работы №5, спроектировать и разработать аллокатор 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).

Описание программы

Исходный код лежит в 13 файлах:

1. main.cpp: основная программа, взаимодействие с пользователем посредством команд из меню
2. tqueueitem.h: описание класса предмета очереди
3. point.h: описание класса точки
4. tqueue.h: описание класса очереди
5. triangle.h: описание класса прямоугольника, наследующегося от figures
6. point.cpp: реализация класса точки
7. tqueue.inl: реализация класса очереди
8. triangle.cpp: реализация класса прямоугольника
9. tqueueitem.inl: реализация класса предмета очереди
10. vector.h
11. titerator.h
12. tallocationblock.cpp
13. tallocationblock.h

Дневник отладки

Пример работы программы:
peter1811@DESKTOP-V53N291: CMakeLists.txt
peter1811@DESKTOP-V53N291: make
peter1811@DESKTOP-V53N291: ./6_lab
0 OK
1 OK
2 OK
3 OK
4 OK
5 OK
6 OK
7 OK
8 OK
9 OK
TAllocationBlock: Memory init
10
TAllocationBlock: Allocate 1 of 10
a1 pointer value:1
9
TAllocationBlock: Allocate 2 of 10
a2 pointer value:2
8
TAllocationBlock: Allocate 3 of 10
a3 pointer value:3
TAllocationBlock: Deallocate block
TAllocationBlock: Deallocate block
7
TAllocationBlock: Allocate 2 of 10
a4 pointer value:4
6
TAllocationBlock: Allocate 3 of 10
a5 pointer value:5
a1 pointer value:1
a2 pointer value:2
a3 pointer value:3
TAllocationBlock: Deallocate block
TAllocationBlock: Deallocate block
TAllocationBlock: Deallocate block
TAllocationBlock: Memory freed
Triangle with vertices (0, 0), (0, 1), (1, 1) was created

```
Queue item: created
Default Triangle is created
Queue item: created
Triangle: (0, 0), (0, 1), (1, 1)
Triangle: (0, 0), (0, 0), (0, 0)
Triangle: (0, 0), (0, 1), (1, 1)
Triangle was deleted
Triangle: (0, 0), (0, 0), (0, 0)
Triangle was deleted
```

Недочёты

Выводы

Я научился работать с аллокаторами данных.

Исходный код

tqueue.h

```
#ifndef TQUEUE_H
#define TQUEUE_H

#include "titerator.h"
#include "tqueueitem.h"

using namespace std;

template <class T> class TQueue {
public:
  TQueue();
  TQueue(const TQueue<T> &other);
  ~TQueue() = default;
  template <class A>
  friend ostream &operator<<(ostream &os, const TQueue<A> &q);
  bool push(shared_ptr<T> &&item);
  bool pop();
  shared_ptr<T> top();
  bool empty();
  size_t size();
  Titerator<TQueue_item<T>, T> begin();
  Titerator<TQueue_item<T>, T> end();

private:
  shared_ptr<TQueue_item<T>> first;
  shared_ptr<TQueue_item<T>> last;
  size_t n;
};
#include "tqueue.inl"

#endif // TQUEUE_H
```

tqueue.inl

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

using namespace std;

template <class T> TQueue<T>::TQueue() : first(nullptr), last(nullptr), n(0) {}

template <class T> TQueue<T>::TQueue(const TQueue<T> &other) {
  first = other.first;
  last = other.last;
  n = other.n;
  cout << "Queue was copied" << endl;
}

template <class T> bool TQueue<T>::push(shared_ptr<T> &&item) {
  shared_ptr<TQueue_item<T>> tail =
      make_shared<TQueue_item<T>>(TQueue_item<T>(item));
  if (tail == nullptr) {
    return false;
  }
  if (this->empty()) { // если пустая очередь, то голова и хвост - один и тот же
                       // элемент
    this->first = this->last = tail;
  } else if (n == 1) { // хвост - вставляемый элемент, а также следующий элемент
                       // от первого
    last = tail;
    first->SetNext(tail);
  } else {
    this->last->SetNext(tail); // хвост - следующий элемент от последнего
    last = tail;
  }
  n++;
  return true;
}

template <class T> bool TQueue<T>::pop() {
  if (first) {
    first = first->GetNext();
    return true;
  }
  return false;
```

```
}

template <class T> shared_ptr<T> TQueue<T>::top() {
  if (first) {
    return first->GetItem();
  }
}

template <class T> size_t TQueue<T>::size() { return n; }

template <class T> bool TQueue<T>::empty() { return first == nullptr; }

template <class T> ostream &operator<<(ostream &os, const TQueue<T> &q) {
  shared_ptr<TQueue_item<T>> item = q.first;
  while (item) {
    os << *item;
    item = item->GetNext();
  }
  return os;
}

template <class T> Titerator<TQueue_item<T>, T> TQueue<T>::begin() {
  return Titerator<TQueue_item<T>, T>(first);
}

template <class T> Titerator<TQueue_item<T>, T> TQueue<T>::end() {
  return Titerator<TQueue_item<T>, T>(nullptr);
}
```

tqueueitem.h

```
#ifndef INC_4_LAB_QUQUE_ITEM_H
#define INC_4_LAB_QUQUE_ITEM_H

#include "triangle.h"
#include <memory>

using namespace std;

template <typename T> class TQueue_item {
public:
  TQueue_item() = default;
  TQueue_item(const shared_ptr<T> &item);
  TQueue_item(const shared_ptr<TQueue_item<T>> &other);
  ~TQueue_item() = default;

  shared_ptr<TQueue_item<T>> SetNext(shared_ptr<TQueue_item<T>> &next_);
  shared_ptr<TQueue_item<T>> GetNext();
  shared_ptr<T> GetItem();

  template <typename A>
  friend ostream &operator<<(ostream &os, const TQueue_item<A> &obj);

private:
  shared_ptr<T> item;
  shared_ptr<TQueue_item<T>> next;
};
#include "tqueueitem.inl"
#endif // INC_4_LAB_QUQUE_ITEM_H
```

tqueueitem.inl

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

using namespace std;

template <class T> TQueue_item<T>::TQueue_item(const shared_ptr<T> &rectangle) {
  this->item = rectangle;
  this->next = nullptr;
  cout << "Queue item: created" << endl;
}

template <class T>
TQueue_item<T>::TQueue_item(const shared_ptr<TQueue_item<T>> &other) {
  this->item = other->item;
  this->next = other->next;
  cout << "Queue item: copied" << endl;
}

template <class T>
shared_ptr<TQueue_item<T>>
TQueue_item<T>::SetNext(shared_ptr<TQueue_item<T>> &next_) {
  shared_ptr<TQueue_item<T>> prev = this->next;
  this->next = next_;
  return prev;
}

template <class T> shared_ptr<T> TQueue_item<T>::GetItem() {
  return this->item;
}

template <class T> shared_ptr<TQueue_item<T>> TQueue_item<T>::GetNext() {
  return this->next;
}

template <class T>
ostream &
operator<<(ostream &os,
           const TQueue_item<T> &obj) { // перегруженный оператор вывода
  if (obj.item) {
    os << "{";
    os << *(obj.item);
```

```
    os << "}" << endl;
  }
  return os;
}
```

point.h

```
#ifndef POINT_H
#define POINT_H

#include <iostream>

class Point {
public:
    Point();

    Point(double x, double y);

    Point &operator++();

    friend Point operator+(const Point &left, const Point &right);

    double dist(Point &other);

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

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

    friend class Triangle; // Дружественные классы, чтобы были доступны координаты точки

private:
    double y_;
    double x_;
};

#endif // POINT_H
```

point.cpp

```cpp
#include "point.h"

#include <cmath>

Point::Point() : x_(0.0), y_(0.0) {} // стандартный конструктор

Point::Point(double x, double y) : x_(x), y_(y) {} // конструктор через значения

double Point::dist(Point &other) { // расстояние между двумя точками
  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) { // перегруженный оператор >>
  is >> p.x_ >> p.y_;
  return is;
}

std::ostream &operator<<(std::ostream &os,
                         const Point &p) { // перегруженный оператор <<
  os << "(" << p.x_ << ", " << p.y_ << ")";
  return os;
}

Point &Point::operator++() {
  this->x_ += 1;
  this->y_ += 1;
  return *this;
}

Point operator+(const Point &left, const Point &right) {
  return Point(left.x_ + right.x_, left.y_ + right.y_);
}
```

triangle.h

```
#ifndef TRIANGLE_H
#define TRIANGLE_H

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

using namespace std;

class Triangle {
public:
    Triangle();

    Triangle(vector<Point>);

    Triangle(const Triangle& other);

    virtual ~Triangle();

    friend istream &operator>>(istream &is, Triangle &obj); // перегруженный оператор >>

    friend ostream &operator<<(ostream &os, const Triangle &obj);

    Triangle &operator++();

    friend Triangle operator+(const Triangle &left, const Triangle &right);

    Triangle &operator=(const Triangle &right);

    size_t VertexNumbers();

    double Area();

private:
    Point a, b, c;
};


#endif
```

triangle.cpp

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


double Triangle::Area() {
    double first = sqrt((a.x_ - b.x_) * (a.x_ - b.x_) + (a.y_ - b.y_) * (a.y_ - b.y_));
    double second = sqrt((b.x_ - c.x_) * (b.x_ - c.x_) + (b.y_ - c.y_) * (b.y_ - c.y_));
    double third = sqrt((a.x_ - c.x_) * (a.x_ - c.x_) + (a.y_ - c.y_) * (a.y_ - c.y_));
    double p = (first + second + third) / 2;
    return sqrt(p * (p - first) * (p - second) * (p - third));
}

Triangle::Triangle() : a(), b(), c(){
    cout << "Default Triangle is created" << endl;
}

Triangle::Triangle(const Triangle &other) {
    a = other.a;
    b = other.b;
    c = other.c;
    cout << "Made copy of Triangle" << endl;
}

Triangle::Triangle(vector<Point> v) : a(v[0]), b(v[1]), c(v[2]) {
    cout << "Triangle with vertices " << a << ", " << b << ", " << c << " was created" <
}

istream &operator>>(istream &is,  Triangle &obj) {
    cout << "Enter cords" << endl;
    is >> obj.a >> obj.b >> obj.c;
    return is;
}

ostream &operator<<(ostream &os, const Triangle &obj) {
    os << "Triangle: " << obj.a << ", " << obj.b << ", " << obj.c;
    return os;
}

Triangle& Triangle::operator++() { // инкрементируем каждую вершину
    ++this->a;
    ++this->b;
```

```
    ++this->c;
    return *this;
}

Triangle operator+(const Triangle &left, const Triangle &right) {
    vector<Point> v{left.a + right.a, left.b + right.b, left.c + right.c};
    return Triangle(v);
}

Triangle& Triangle::operator=(const Triangle &other) {
    if (this == &other){
        return *this;
    }
    this->a = other.a;
    this->b = other.b;
    this->c = other.c;
    return *this;
}

Triangle::~Triangle() {
    cout << "Triangle was deleted" << endl;
}

size_t Triangle::VertexNumbers() {
    return 3;
}
```

main.cpp

```cpp
#include <iostream>
#include <memory>

#include "triangle.h"
#include "tqueue.h"
#include <vector>
#include "tallocationblock.h"

void TestQueue() {
    TQueue<Triangle> queue;
    vector<Point> v;
    v.emplace_back(0, 0);
    v.emplace_back(0, 1);
    v.emplace_back(1, 1);
    queue.push(make_shared<Triangle>(v));
    queue.push(make_shared<Triangle>());

    for (auto i: queue) {
        std::cout << *i << std::endl;
    }

    while (!queue.empty()) {
        std::cout << *queue.top() << std::endl;
        queue.pop();
    }
}

void TestAllocationBlock() {
    TAllocationBlock allocator(sizeof(int), 10);
    int *a1 = nullptr;
    int *a2 = nullptr;
    int *a3 = nullptr;
    int *a4 = nullptr;
    int *a5 = nullptr;

    a1 = (int *) allocator.allocate();
    *a1 = 1;
    std::cout << "a1 pointer value:" << *a1 << std::endl;

    a2 = (int *) allocator.allocate();
    *a2 = 2;
```

```
    std::cout << "a2 pointer value:" << *a2 << std::endl;

    a3 = (int *) allocator.allocate();
    *a3 = 3;
    std::cout << "a3 pointer value:" << *a3 << std::endl;

    allocator.deallocate(a1);
    allocator.deallocate(a3);

    a4 = (int *) allocator.allocate();
    *a4 = 4;
    std::cout << "a4 pointer value:" << *a4 << std::endl;

    a5 = (int *) allocator.allocate();
    *a5 = 5;
    std::cout << "a5 pointer value:" << *a5 << std::endl;

    std::cout << "a1 pointer value:" << *a1 << std::endl;
    std::cout << "a2 pointer value:" << *a2 << std::endl;
    std::cout << "a3 pointer value:" << *a3 << std::endl;

    allocator.deallocate(a2);
    allocator.deallocate(a4);
    allocator.deallocate(a5);
}

int main(int argc, char **argv) {
    TestAllocationBlock();
    TestQueue();
    return 0;
}
```

tallocationblock.cpp

```cpp
#include "tallocationblock.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) {
        vec_free_blocks.push_back(_used_blocks + i * _size);
        std::cout << i << " OK" << std::endl;
    }
    _free_count = _count;
    std::cout << "TAllocationBlock: Memory init" << std::endl;
}

void *TAllocationBlock::allocate() {
    void *result = nullptr;

    if (_free_count > 0) {
        std::cout << vec_free_blocks.size() << std::endl;
        result = vec_free_blocks.back();
        vec_free_blocks.pop();
        _free_count--;
        std::cout << "TAllocationBlock: Allocate " << (_count - _free_count);
        std::cout << " of " << _count << std::endl;

    } else {
        std::cout << "TAllocationBlock: No memory exception :-)" << std::endl;
    }

    return result;
}

void TAllocationBlock::deallocate(void *pointer) {
    std::cout << "TAllocationBlock: Deallocate block " << std::endl;

    vec_free_blocks[_free_count] = pointer;
    _free_count++;
}

bool TAllocationBlock::has_free_blocks() {
    return _free_count > 0;
```

```
}

TAllocationBlock::~TAllocationBlock() {
    if (_free_count < _count) {
        std::cout << "TAllocationBlock: Memory leak?" << std::endl;
    } else {
        std::cout << "TAllocationBlock: Memory freed" << std::endl;
    }
    delete _used_blocks;
}
```

tallocationblock.h

```
#ifndef TALLOCATION_BLOCK_H
#define TALLOCATION_BLOCK_H

#include "vector.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;
  Vector<void*> vec_free_blocks;
  size_t _free_count;
};

#endif  // TALLOCATION_BLOCK_H
```

vector.h

```
#ifndef DATA_VECTOR_H
#define DATA_VECTOR_H

#include <iostream>

template<typename T>
class Vector {
public:
    Vector() {
        arr_ = new T[1];
        capacity_ = 1;
    }

    Vector(Vector &other) {
        if (this != &other) {
            delete[] arr_;
            arr_ = other.arr_;
            size_ = other.size_;
            capacity_ = other.capacity_;
            other.arr_ = nullptr;
            other.size_ = other.capacity_ = 0;
        }
    }

    Vector(Vector &&other) noexcept {
        if (this != &other) {
            delete[] arr_;
            arr_ = other.arr_;
            size_ = other.size_;
            capacity_ = other.capacity_;
            other.arr_ = nullptr;
            other.size_ = other.capacity_ = 0;
        }
    }

    Vector &operator=(Vector &other) {
        if (this != &other) {
            delete[] arr_;
            arr_ = other.arr_;
            size_ = other.size_;
            capacity_ = other.capacity_;
```

```
            other.arr_ = nullptr;
            other.size_ = other.capacity_ = 0;
        }
        return *this;
    }

    Vector &operator=(Vector &&other) noexcept {
        if (this != &other) {
            delete[] arr_;
            arr_ = other.arr_;
            size_ = other.size_;
            capacity_ = other.capacity_;
            other.arr_ = nullptr;
            other.size_ = other.capacity_ = 0;
        }
        return *this;
    }

    ~Vector() {
        delete[] arr_;
    }

public:
    [[nodiscard]] bool isEmpty() const {
        return size_ == 0;
    }

    [[nodiscard]] size_t size() const {
        return size_;
    }

    [[nodiscard]] size_t capacity() const {
        return capacity_;
    }

    void push_back(const T &value) {
        if (size_ >= capacity_) addMemory();
        arr_[size_++] = value;
    }

    void pop() {
        --size_;
```

```
    }

    T &back() {
        return arr_[size_ - 1];
    }

    void remove(size_t index) {
        for (size_t i = index + 1; i < size_; ++i) {
            arr_[i - 1] = arr_[i];
        }
        --size_;
    }

public:
    T *begin() {
        return &arr_[0];
    }

    const T *begin() const {
        return &arr_[0];
    }

    T *end() {
        return &arr_[size_];
    }

    const T *end() const {
        return &arr_[size_];
    }

public:
    T &operator[](size_t index) {
        return arr_[index];
    }

    const T &operator[](size_t index) const {
        return arr_[index];
    }

private:

    void addMemory() {
```

```
        capacity_ *= 2;
        T *tmp = arr_;
        arr_ = new T[capacity_];
        for (size_t i = 0; i < size_; ++i) arr_[i] = tmp[i];
        delete[] tmp;
    }

    T *arr_;
    size_t size_{};
    size_t capacity_{};
};

template<typename T>
inline std::ostream &operator<<(std::ostream &os, const Vector<T> &vec) {
    for (const T &val: vec) os << val << " ";
    return os;
}

#endif
```