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

ЛАБОРАТОРНАЯ РАБОТА №8
по курсу “Объектно-ориентированное программирование”
III семестр, 2021/22 учебный год

Выполнила студентка группы *М8О-208Б-20*
Шатунова Юлия Викторовна
Преподаватель: *Дорохов Евгений Павлович*

Москва, 2021

Цель работы

Используя структуру данных, разработанную для лабораторной работы №7, спроектировать и разработать аллокатор памяти для динамической структуры данных. Целью построения аллокатора является минимизация вызова операции malloc.

Задание

Аллокатор должен выделять большие блоки памяти для хранения фигур и при создании новых фигур-объектов выделять место под объекты в этой памяти.

Аллокатор 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 операторы new и delete у классов-фигур.

Вариант №26. Фигура – квадрат (Square), контейнер первого уровня – очередь (TQueue).

Описание программы

Исходный код разделён на 14 файлов:

- figure.h – описание класса фигуры
- point.h – описание класса точки
- point.cpp – реализация класса точки
- square.h – описание класса квадрата
- square.cpp – реализация класса квадрата
- tqueue_item.h – описание элемента очереди
- tqueue_item.cpp – реализация элемента очереди
- tqueue.h – описание очереди
- tqueue.cpp – реализация очереди
- main.cpp – основная программа
- iterator.h – реализация итератора по очереди
- tallocation_block.h – описание аллокатора
- tallocation_block.cpp – реализация аллокатора
- vector.h – пользовательское описание вектора

Дневник отладки

При выполнении работы отладка не требовалась.

Недочеты

Недочеты не были обнаружены.

Выводы

В ходе выполнения лабораторной работы №8 был реализован аллокатор (распределитель памяти) на языке программирования C++. Это специализированный класс, реализующий и инкапсулирующий малозначимые (с прикладной точки зрения) детали распределения и освобождения ресурсов компьютерной памяти.

Исходный код

figure.h

```
#ifndef FIGURE_H
#define FIGURE_H

#include "point.h"

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

#endif // FIGURE_H
```

main.cpp

```
#include "tallocation_block.h"
#include "tqueue.h"

void _Queue() {
    TQueue<Square> queue;
    std::vector<Point> vect;
    Point a_1(1.0, 1.0);
    Point b_1(1.0, 2.0);
    Point c_1(2.0, 2.0);
    Point d_1(2.0, 1.0);
    Point a_2(3.0, 1.0);
    Point b_2(3.0, 3.0);
    Point c_2(5.0, 3.0);
    Point d_2(5.0, 1.0);
    Point a_3(0.0, 0.0);
    Point b_3(0.0, 4.0);
    Point c_3(4.0, 4.0);
    Point d_3(4.0, 0.0);
    queue.Push(std::shared_ptr<Square>(new Square(a_1, b_1, c_1, d_1)));
    queue.Push(std::shared_ptr<Square>(new Square(a_2, b_2, c_2, c_2)));
    queue.Push(std::shared_ptr<Square>(new Square(a_3, b_3, c_3, d_3)));

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

    while (!queue.Empty()) {
        std::cout << *queue.Top() << std::endl;
        queue.Pop();
    }
}

void _AllocationBlock() {
    TAllocationBlock allocator(sizeof(int), 10);
    int* alloc1 = nullptr;
    int* alloc2 = nullptr;
    int* alloc3 = nullptr;
    int* alloc4 = nullptr;
    int* alloc5 = nullptr;



    alloc1 = (int*)allocator.allocate();
    *alloc1 = 1;
    std::cout << "a1 pointer value:" << *alloc1 << std::endl;

    alloc2 = (int*)allocator.allocate();
    *alloc2 = 2;
    std::cout << "a2 pointer value:" << *alloc2 << std::endl;

    alloc3 = (int*)allocator.allocate();
    *alloc3 = 3;
    std::cout << "a3 pointer value:" << *alloc3 << std::endl;

    allocator.deallocate(alloc1);
    allocator.deallocate(alloc3);

    alloc4 = (int*)allocator.allocate();
    *alloc4 = 4;
    std::cout << "a4 pointer value:" << *alloc4 << std::endl;

    alloc4 = (int*)allocator.allocate();
    *alloc4 = 5;
    std::cout << "a5 pointer value:" << *alloc4 << std::endl;

    std::cout << "a1 pointer value:" << *alloc1 << std::endl;
    std::cout << "a2 pointer value:" << *alloc2 << std::endl;
    std::cout << "a3 pointer value:" << *alloc3 << std::endl;

    allocator.deallocate(alloc2);
    allocator.deallocate(alloc4);
    allocator.deallocate(alloc5);
```

```
}

int main(int argc, char** argv) {
    _AllocationBlock();
    _Queue();
    return 0;
}
```

point.cpp

```
#include "point.h"

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 operator+(Point x, Point y) {
    return Point(x.x_ + y.x_, x.y_ + y.y_);
}
```

point.h

```
#ifndef POINT_H
#define POINT_H

#include <iostream>
#include <ostream>
#include <vector>
#include <cmath>
```

```
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);
    friend Point operator+(Point a, Point b);

    friend class Square;
    friend class Rectangle;
    friend class Trapezoid;

private:
    double x_;
    double y_;
};

#endif // POINT_H
```

square.cpp

```
#include "square.h"

Square::Square() : point_a(0.0, 0.0), point_b(0.0, 0.0), point_c(0.0, 0.0), point_d(0.0, 0.0) {
        std::cout << "Default square is created" << std::endl;
}

Square::Square(Point a, Point b, Point c, Point d) : point_a(a), point_b(b), point_c(c), point_d(d) {
        std::cout << "Square is created with vertices: ";
        std::cout << point_a << ", ";
        std::cout << point_b << ", ";
        std::cout << point_c << ", ";
        std::cout << point_d << std::endl;
}

Square::Square(const Square& other) : Square(other.point_a, other.point_b, other.point_c, other.point_d) {
        std::cout << "Square's copy is created" << std::endl;
}

double Square::Area() {
        double side = 0.0;
        double fig_square = 0.0;
        side = point_b.dist(point_a);
        fig_square = side * side;
```

```
    return fig_square;
}

std::istream& operator>>(std::istream& is, Square& obj) {
    is >> obj.point_a >> obj.point_b >> obj.point_c >> obj.point_d;
    return is;
}

std::ostream& operator<<(std::ostream& os, const Square& obj) {
    Point a(obj.point_a);
    Point b(obj.point_b);
    Point c(obj.point_c);
    Point d(obj.point_d);
    os << "Point_a: " << a << ", ";
    os << "Point_b: " << b << ", ";
    os << "Point_c: " << c << ", ";
    os << "Point_d: " << d << std::endl;
    return os;
}

Square& Square::operator++() {
    point_a.x_ += 1.0;
    point_a.y_ += 1.0;
    point_b.x_ += 1.0;
    point_b.y_ += 1.0;
    point_c.x_ += 1.0;
    point_c.y_ += 1.0;
    point_d.x_ += 1.0;
    point_d.y_ += 1.0;

    return *this;
}

Square operator+(const Square& left, const Square& right) {
    return Square(left.point_a + right.point_a, left.point_b + right.point_b, left.point_c + right.point_c, left.point_d + right.point_d);
}

Square& Square::operator=(const Square& other) {
    if (this == &other) {
        return *this;
    }
    else {
        point_a = other.point_a;
        point_b = other.point_b;
        point_c = other.point_c;
        point_d = other.point_d;
        std::cout << "Square is copied" << std::endl;
```

```
                return *this;
        }
}

Square::~Square() {
        std::cout << "Square is deleted" << std::endl;
}
```

square.h

```
#ifndef SQUARE_H
#define SQUARE_H

#include "figure.h"

class Square : public Figure {
public:
        Square();
        Square(Point a, Point b, Point c, Point d);
        Square(const Square& other);

        double Area();

        friend std::istream& operator>>(std::istream& is, Square& obj);
        friend std::ostream& operator<<(std::ostream& os, const Square& obj);

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

        Square& operator=(const Square& other);

        virtual ~Square();

private:
        Point point_a; // lower left corner, then clockwise
        Point point_b;
        Point point_c;
        Point point_d;
};

#endif // SQUARE_H
```

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) {
        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;
}
```

tallocation_block.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
```

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->GetValue();
    }

    std::shared_ptr<T> operator->() {
        return node_ptr->GetValue();
    }

    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 // TITERATOR_H
```

tqueue.cpp

```
#include "tqueue.h"

template <class T>
TQueue<T>::TQueue() : head(nullptr), tail(nullptr), num_of_elem(0) {

}

template <class T>
TQueue<T>::TQueue(const TQueue<T>& other) {
    head = other.head;
}

template <class T>
std::ostream& operator<<(std::ostream& os, const TQueue<T>& queue) {
    std::shared_ptr<TQueueItem<T>> item = queue.head;

    while (item != nullptr) {
        os << *item << " => ";
        item = item->GetNext();
    }
    return os;
}

template <class T>
```

```
void TQueue<T>::Push(std::shared_ptr<T> &&square) {
    std::shared_ptr<TQueueItem<T>> item =
std::make_shared<TQueueItem<T>>(TQueueItem<T>(square));
    if (item != nullptr) {
        if (this->Empty()) {
            this->head = this->tail = item;
        }
        else if (num_of_elem == 1) {
            tail = item;
            head->SetNext(item);
        }
        else {
            this->tail->SetNext(item);
            tail = item;
        }
        num_of_elem++;
    }
}

template <class T>
std::shared_ptr<T> TQueue<T>::Pop() {
    std::shared_ptr<T> result;
    if (head != nullptr) {
        result = head->GetValue();
        head = head->GetNext();
        //item->SetNext(nullptr);
        //delete item;
    }
    return result;
}

template <class T>
std::shared_ptr<T> TQueue<T>::Top() {
    if (head) {
        return head->GetValue();
    }
}

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

template <class T>
size_t TQueue<T>::Length() {
    return num_of_elem;
}
```

```
template <class T>
TIterator<TQueueItem<T>, T> TQueue<T>::begin() {
    return TIterator<TQueueItem<T>, T>(head);
}

template <class T>
TIterator<TQueueItem<T>, T> TQueue<T>::end() {
    return TIterator<TQueueItem<T>, T>(nullptr);
}

template <class T>
TQueue<T>::~TQueue() {

}

#include "square.h"
template class TQueue<Square>;
template std::ostream& operator<<(std::ostream& os, const TQueue<Square>& queue);
```

tqueue.h

```
#ifndef TQUEUE_H
#define TQUEUE_H

#include "titerator.h"
#include "tqueue_item.h"

template <class T>
class TQueue {
public:
    TQueue();
    TQueue(const TQueue<T>& other);

    void Push(std::shared_ptr<T> &&square);
    std::shared_ptr<T> Pop();

    std::shared_ptr<T> Top();

    bool Empty();

    size_t Length();

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

    TIterator<TQueueItem<T>, T> begin();
    TIterator<TQueueItem<T>, T> end();

    virtual ~TQueue();
private:
    std::shared_ptr<TQueueItem<T>> head;
    std::shared_ptr<TQueueItem<T>> tail;
    size_t num_of_elem;
};

#endif // TQUEUE_H
```

tqueue_item.cpp

```
#include "tqueue_item.h"

template <class T>
TQueueItem<T>::TQueueItem(const std::shared_ptr<T>& item):
    item(item), next(nullptr) {
    std::cout << "Queue item is created" << std::endl;
}

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

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

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

template <class T>
std::shared_ptr<T> TQueueItem<T>::GetValue() const {
    return this->item;
}

template <class A>
std::ostream& operator<<(std::ostream& os, const TQueueItem<A>& obj) {
    os << "Item: " << *obj.item << std::endl;
    return os;
}
```

```
template <class T>
void* TQueueItem<T>::operator new(size_t size) {
    std::cout << "Allocated: " << size << " bytes" << std::endl;
    return malloc(size);
}

template <class T>
void TQueueItem<T>::operator delete(void* p) {
    std::cout << "Deleted" << std::endl;
    free(p);
}

template <class T>
TQueueItem<T>::~TQueueItem() {
    std::cout << "The queue item is deleted" << std::endl;
}

#include "square.h"
template class TQueueItem<Square>;
template std::ostream& operator<<(std::ostream& os, const TQueueItem<Square>& obj);
```

tqueue_item.h

```
#ifndef TQUEUE_ITEM_H
#define TQUEUE_ITEM_H

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

template <class T> class TQueueItem {
public:
    TQueueItem(const std::shared_ptr<T>& square);
    TQueueItem(const std::shared_ptr<TQueueItem<T>>& other);

    std::shared_ptr<TQueueItem<T>> SetNext(std::shared_ptr<TQueueItem> &next);
    std::shared_ptr<TQueueItem<T>> GetNext();

    std::shared_ptr<T> GetValue() const;

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

    void* operator new(size_t size);
    void operator delete(void* p);

    virtual ~TQueueItem();

private:
    std::shared_ptr<T> item;
    std::shared_ptr<TQueueItem<T>> next;
};

#endif // TQUEUE_ITEM_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
```