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