МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ  
ФЕДЕРАЦИИ МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

ЛАБОРАТОРНАЯ РАБОТА №8 по курсу объектно-ориентированное программирование I семестр, 2021/22 уч. год

Студент *Зубко Дмитрий Валерьевич, группа М80-208Б-20*

Преподаватель *Дорохов Евгений Павлович*

**Цель работы:**  
Целью лабораторной работы является:   
  
Закрепление навыков по работе с памятью в C++;   
Создание аллокаторов памяти для динамических структур данных.  
  
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. 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 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.

**Дневник отладки**

Во время выполнения лабораторной были некие трудности с реализацией линейного списка и аллокатора, позже они были полностью ликвидирован.

**Недочёты**  
Недочётов не было обнаружено.

**Выводы**

Лабораторная работа №8 позволила мне реализовать свой класс аллокаторов, полностью прочувствовать процесс выделения памяти на низкоуровневых языках программирования. Лабораторная прошла успешно. В процессе выполнения работы я на практике познакомился с понятием аллокатора. Так как во многих структурах данных используются аллокаторы, то это очень важная тема, которую должен знать каждый программист на С++. Написание собственноручного итератора помогает реализовать собственную логику выделения памяти, которая может быть более оправданной в некоторых ситуациях, чем стандартный аллокатор, как для самописных, так и для стандартных структур данных.

**Исходный код**

figure.h

#ifndef OOP5\_FIGURE\_H

#define OOP5\_FIGURE\_H

#include <cmath>

#include <iostream>

#include "point.h"

class Figure {

public:

virtual size\_t VertexesNumber() = 0;

virtual double Area() = 0;

virtual void Print(std::ostream &os) = 0;

virtual ~Figure() {};

};

#endif //OOP5\_FIGURE\_H

main.cpp  
  
#include "pentagon.h"

#include "TVector.h"

#include "TStack.h"

int main() {

TVector<Pentagon> t;

t.InsertLast(std::shared\_ptr<Pentagon>(new Pentagon(

{1.f, 9.f}, {8.f, 7.f}, {6.f, 5.f}, {4.f, 3.f}, {2.f, 1.f})));

t.InsertLast(std::shared\_ptr<Pentagon>(new Pentagon(

{1.f, 9.f}, {8.f, 7.f}, {6.f, 5.f}, {4.f, 3.f}, {2.f, 1.f})));

t.InsertLast(std::shared\_ptr<Pentagon>(new Pentagon(

{1.f, 9.f}, {8.f, 7.f}, {6.f, 5.f}, {4.f, 3.f}, {2.f, 1.f})));

std::cout << t << std::endl;

for (auto i = t.begin(); i != t.end(); ++i)

(\*i).Print(std::cout);

}

pentagon.cpp  
  
#include "pentagon.h"

TAllocationBlock Pentagon::\_alloc\_block(sizeof(Pentagon), 32);

std::istream& operator>>(std::istream& is, Pentagon& p) {

std::cout << "Enter data:" << std::endl;

is >> p.a >> p.b >> p.c >> p.d >> p.e;

// std::cout << "Pentagon created via istream" << std::endl;

return is;

}

std::ostream& operator<<(std::ostream& os, Pentagon& p) {

os << "Pentagon: " << p.a << p.b << p.c << p.d << p.e << std::endl;

return os;

}

Pentagon& Pentagon::operator=(const Pentagon &other) {

this->a = other.a;

this->b = other.b;

this->c = other.c;

this->d = other.d;

this->e = other.e;

return \*this;

}

bool Pentagon::operator==(const Pentagon &other) {

return a == other.a && b == other.b && c == other.c && d == other.d && e == other.e;

}

size\_t Pentagon::VertexesNumber() {

return 5;

}

double Pentagon::SquareTriangle(Point a, Point b, Point c){

double p = (a.dist(b) + b.dist(c) + c.dist(a)) / 2;

return sqrt(p \* (p - a.dist(b)) \* (p - b.dist(c)) \* (p - c.dist(a)));

}

double Pentagon::Area() {

return SquareTriangle(a, b, c) + SquareTriangle(a, c, d) + SquareTriangle(a, d, e);

}

void Pentagon::Print(std::ostream &os) {

os << "Pentagon: " << a << b << c << d << e << std::endl;

}

Pentagon::Pentagon(){}

Pentagon::Pentagon(Point a\_, Point b\_, Point c\_, Point d\_, Point e\_) : a(a\_), b(b\_), c(c\_), d(d\_), e(e\_) {}

Pentagon::Pentagon(const Pentagon &other) : Pentagon(other.a, other.b, other.c, other.d, other.e) {

}

Pentagon::Pentagon(std::istream &is) {

std::cout << "Enter data:" << std::endl;

is >> a >> b >> c >> d >> e;

// std::cout << "Pentagon created via istream" << std::endl;

}

Pentagon.h  
  
#ifndef OOP5\_PENTAGON\_H

#define OOP5\_PENTAGON\_H

#include "figure.h"

#include "tallocation.h"

class Pentagon : Figure{

public:

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

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

size\_t VertexesNumber() override;

double Area() override;

void Print(std::ostream &os) override;

bool operator==(const Pentagon& other);

Pentagon();

Pentagon(Point a\_, Point b\_, Point c\_, Point d\_, Point e\_);

Pentagon(std::istream &is);

Pentagon(const Pentagon &other);

Pentagon& operator=(const Pentagon& other);

void \*operator new(size\_t size)

{

return \_alloc\_block.Allocate(size);

}

void operator delete(void \*pointer)

{

\_alloc\_block.Free(pointer);

}

private:

Point a, b, c, d, e;

double SquareTriangle(Point a, Point b, Point c);

static TAllocationBlock \_alloc\_block;

};

#endif //OOP5\_PENTAGON\_H

Point.cpp

#include "point.h"

#include <cmath>

bool Point::operator==(const Point &other) {

return (this->x\_ == other.x\_ && this->y\_ == other.y\_);

}

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 OOP5\_POINT\_H

#define OOP5\_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);

bool operator==(const Point& other);

private:

double x\_;

double y\_;

};

#endif //OOP5\_POINT\_H

TVector.cpp

#pragma once

#include <iostream>

#include <memory>

#include <cstdlib>

#include "TIterator.h"

template <typename T>

class TVector {

public:

TVector();

TVector(const TVector &);

virtual ~TVector();

size\_t Length() const {

return length\_;

}

bool Empty() const {

return !length\_;

}

const std::shared\_ptr<T> &operator[](const size\_t index) const {

return data\_[index];

}

std::shared\_ptr<T> &Last() const {

return data\_[length\_ - 1];

}

void InsertLast(const std::shared\_ptr<T> &);

void EmplaceLast(const T &&);

void Remove(const size\_t index);

T RemoveLast() {

return \*data\_[--length\_];

}

void Clear();

TIterator<T> begin() {

return TIterator<T>(data\_);

}

TIterator<T> end() {

return TIterator<T>(data\_ + length\_);

}

template<typename TF>

friend std::ostream &operator<<(

std::ostream &, const TVector<TF> &);

private:

void \_Resize(const size\_t new\_capacity);

std::shared\_ptr<T> \*data\_;

size\_t length\_, capacity\_;

};

template <typename T>

TVector<T>::TVector()

: data\_(new std::shared\_ptr<T>[32]),

length\_(0), capacity\_(32) {}

template <typename T>

TVector<T>::TVector(const TVector &vector)

: data\_(new std::shared\_ptr<T>[vector.capacity\_]),

length\_(vector.length\_), capacity\_(vector.capacity\_) {

std::copy(vector.data\_, vector.data\_ + vector.length\_, data\_);

}

template <typename T>

TVector<T>::~TVector() {

delete[] data\_;

}

template <typename T>

void TVector<T>::\_Resize(const size\_t new\_capacity) {

std::shared\_ptr<T> \*newdata = new std::shared\_ptr<T>[new\_capacity];

std::copy(data\_, data\_ + capacity\_, newdata);

delete[] data\_;

data\_ = newdata;

capacity\_ = new\_capacity;

}

template <typename T>

void TVector<T>::InsertLast(const std::shared\_ptr<T> &item) {

if (length\_ >= capacity\_)

\_Resize(capacity\_ << 1);

data\_[length\_++] = item;

}

template <typename T>

void TVector<T>::EmplaceLast(const T &&item) {

if (length\_ >= capacity\_)

\_Resize(capacity\_ << 1);

data\_[length\_++] = std::make\_shared<T>(item);

}

template <typename T>

void TVector<T>::Remove(const size\_t index) {

std::copy(data\_ + index + 1, data\_ + length\_, data\_ + index);

--length\_;

}

template <typename T>

void TVector<T>::Clear() {

delete[] data\_;

data\_ = new std::shared\_ptr<T>[32];

length\_ = 0;

capacity\_ = 32;

}

template <typename T>

std::ostream &operator<<(std::ostream &os, const TVector<T> &vector) {

const size\_t last = vector.length\_ - 1;

for (size\_t i = 0; i < vector.length\_; ++i)

os << (\*vector.data\_[i]);

os << std::endl;

return os;

}

TIterator.h  
  
#pragma once

#include <memory>

template <typename T>

class TIterator {

public:

TIterator(std::shared\_ptr<T> \*iter) : iter\_(iter) {}

T operator\*() const {

return \*(\*iter\_);

}

T operator->() const {

return \*(\*iter\_);

}

void operator++() {

iter\_ += 1;

}

TIterator operator++(int) {

TIterator iter(\*this);

++(\*this);

return iter;

}

bool operator==(TIterator const &iterator) const {

return iter\_ == iterator.iter\_;

}

bool operator!=(TIterator const &iterator) const {

return iter\_ != iterator.iter\_;

}

private:

std::shared\_ptr<T> \*iter\_;

};

tallocation.h  
  
#ifndef OOP6\_TALLOCATION\_H

#define OOP6\_TALLOCATION\_H

#include "tstack.h"

class TAllocationBlock {

public:

TAllocationBlock(size\_t size, size\_t count);

~TAllocationBlock();

void \*Allocate(size\_t size);

void Free(void \*pointer);

inline bool FreeBlocksAvailable() const {

return budget\_;

}

private:

void \_Resize(size\_t new\_count);

size\_t size\_;

size\_t count\_;

size\_t budget\_;

char \*used\_blocks\_;

TStack<void \*> free\_blocks\_;

};

#endif //OOP6\_TALLOCATION\_H

tallocation.cpp

#include "tallocation.h"

#include <iostream>

TAllocationBlock::TAllocationBlock(size\_t size, size\_t count)

: size\_(size), count\_(count), budget\_(count),

used\_blocks\_(new char[size \* count])

{

for (size\_t i = 0; i < count; ++i)

free\_blocks\_.Emplace((void \*)(used\_blocks\_ + (i \* size)));

}

TAllocationBlock::~TAllocationBlock()

{

delete[] used\_blocks\_;

}

void TAllocationBlock::\_Resize(size\_t new\_count)

{

char \*newdata = new char[size\_ \* new\_count];

std::copy(used\_blocks\_, used\_blocks\_ + (size\_ \* count\_), newdata);

delete[] used\_blocks\_;

used\_blocks\_ = newdata;

count\_ = new\_count;

}

void \*TAllocationBlock::Allocate(size\_t size)

{

if (size != size\_) {

std::cerr << "This block allocates " << size\_ << "bytes only. "

<< "You tried to allocate " << size << '\n';

return 0;

}

if (!budget\_) {

size\_t old\_cound = count\_;

\_Resize(count\_ << 1);

budget\_ += (count\_ - old\_cound);

for (size\_t i = old\_cound; i < count\_; ++i)

free\_blocks\_.Emplace((void \*)(used\_blocks\_ + (i \* size\_)));

}

--budget\_;

return free\_blocks\_.Pop();

}

void TAllocationBlock::Free(void \*pointer)

{

free\_blocks\_.Push(std::make\_shared<void \*>(pointer));

++budget\_;

}

TStack.h  
  
#pragma once

#include <ostream>

#include <memory>

#include <cstdlib>

template <typename T>

class TStack {

public:

TStack();

TStack(const TStack &);

virtual ~TStack();

size\_t Length() const {

return length\_;

}

bool Empty() const {

return !length\_;

}

std::shared\_ptr<T> &Top() const {

return data\_[length\_ - 1];

}

void Emplace(const T &&);

void Push(const std::shared\_ptr<T> &);

inline T Pop() {

return \*data\_[--length\_];

}

void Clear();

template<typename TF>

friend std::ostream &operator<<(

std::ostream &, const TStack<TF> &);

private:

void \_Resize(const size\_t new\_capacity);

std::shared\_ptr<T> \*data\_;

size\_t length\_, capacity\_;

};

template <typename T>

TStack<T>::TStack()

: data\_(new std::shared\_ptr<T>[32]),

length\_(0), capacity\_(32) {}

template <typename T>

TStack<T>::TStack(const TStack &vector)

: data\_(new std::shared\_ptr<T>[vector.capacity\_]),

length\_(vector.length\_), capacity\_(vector.capacity\_) {

std::copy(vector.data\_, vector.data\_ + vector.length\_, data\_);

}

template <typename T>

TStack<T>::~TStack() {

delete[] data\_;

}

template <typename T>

void TStack<T>::\_Resize(const size\_t new\_capacity) {

std::shared\_ptr<T> \*newdata = new std::shared\_ptr<T>[new\_capacity];

std::copy(data\_, data\_ + capacity\_, newdata);

delete[] data\_;

data\_ = newdata;

capacity\_ = new\_capacity;

}

template <typename T>

void TStack<T>::Emplace(const T &&item)

{

if (length\_ >= capacity\_)

\_Resize(capacity\_ << 1);

data\_[length\_++] = std::make\_shared<T>(item);

}

template <typename T>

void TStack<T>::Push(const std::shared\_ptr<T> &item) {

if (length\_ >= capacity\_)

\_Resize(capacity\_ << 1);

data\_[length\_++] = item;

}

template <typename T>

void TStack<T>::Clear() {

delete[] data\_;

data\_ = new std::shared\_ptr<T>[32];

length\_ = 0;

capacity\_ = 32;

}

template <typename T>

std::ostream &operator<<(std::ostream &os, const TStack<T> &stack) {

for (size\_t i = stack.length\_ - 1; i >= 0; --i)

os << (\*stack.data\_[i]);

return os;

}