МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ  
ФЕДЕРАЦИИ МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

ЛАБОРАТОРНАЯ РАБОТА №6 по курсу объектно-ориентированное программирование I семестр, 2021/22 уч. год

Студент *Васютинский Вадим Александрович, 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**

Данная лабораторная работа превратилась для меня в личный ад. Каждая написанная мной реализация работала в определённых случаях, но в определённый момент вылетала. Итоговый вариант, есть сплав всех неудачных версий.

**Недочёты**  
Из-за недостаточно продолжительной отладки я могу сомневаться в абсолютно безопасной работе программы, но на проведённых мной тестах программа выдавала правильный результат.

**Выводы**

Лабораторная работа №6 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 аллокатора 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 "rectangle.h"

#include "TVector.h"

#include <memory>

#include <string>

int main() {

std::string command;

TVector<Rectangle> v;

while (std::cin >> command) {

if (command == "print")

std::cout << v;

else if (command == "insertlast") {

Rectangle r;

std::cin >> r;

std::shared\_ptr<Rectangle> d(new Rectangle(r));

v.InsertLast(d);

}

else if (command == "removelast") {

v.RemoveLast();

}

else if (command == "last") {

std::cout << v.Last();

}

else if (command == "idx") {

int idx;

std::cin >> idx;

std::cout << v[idx];

}

else if (command == "clear") {

v.Clear();

}

else if (command == "empty") {

if (v.Empty()) std::cout << "Yes" << std::endl;

else std::cout << "No" << std::endl;

}

else if (command == "printall") {

for (auto rect: v) {

std::cout << rect;

}

std::cout << std::endl;

}

}

return 0;

}

rectangle.cpp  
  
#include "rectangle.h"

std::istream& operator>>(std::istream& is, Rectangle& r) {

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

is >> r.a >> r.b >> r.c >> r.d;

return is;

}

std::ostream& operator<<(std::ostream& os, Rectangle& r) {

os << "Pentagon: " << r.a << r.b << r.c << r.d;

return os;

}

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

this->a = other.a;

this->b = other.b;

this->c = other.c;

this->d = other.d;

return \*this;

}

rectangle.h  
  
#ifndef OOP1\_RECTANGLE\_H

#define OOP1\_RECTANGLE\_H

#include "figure.h"

class Rectangle : Figure {

public:

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

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

bool operator==(const Rectangle& other);

Rectangle& operator=(const Rectangle& other);

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

size\_t VertexesNumber() override;

double Area() override;

Rectangle();

Rectangle(Point a, Point b, Point c, Point d);

Rectangle(std::istream &is);

Rectangle(const Rectangle &other);

virtual ~Rectangle();

private:

Point a, b, c, d;

};

#endif //OOP1\_RECTANGLE\_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

//

// Created by queens\_gambit on 16.01.2022.

//

#include "TVector.h"

#include "rectangle.h"

#include <cassert>

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) {

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

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

os << std::endl;

return os;

}

template class TVector<Rectangle>;

template std::ostream& operator<<(std::ostream& os, const TVector<Rectangle >& arr);

TIterator.h  
  
#ifndef OOP5\_TITERATOR\_H

#define OOP5\_TITERATOR\_H

#include <memory>

#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\_;

};

#endif //OOP5\_TITERATOR\_H

TAllocationBlock.h  
  
#ifndef OOP6\_TALLOCATIONBLOCK\_H

#define OOP6\_TALLOCATIONBLOCK\_H

#include "TStack.h"

class TAllocationBlock {

public:

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

~TAllocationBlock();

void \*Allocate(size\_t size);

void Free(void \*pointer);

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\_TALLOCATIONBLOCK\_H

TAllocationBlock.cpp

#include "TAllocationBlock.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 != this->size) {

std::cerr << "This block allocates " << this->size << "bytes only. "

<< "You tried to allocate " << size << std::endl;

return 0;

}

if (!budget) {

size\_t old\_count = count;

Resize(count << 1);

budget += (count - old\_count);

for (size\_t i = old\_count; 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  
  
#include <memory>

#include <cstdlib>

template <typename T>

class TStack {

public:

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];

}

void Emplace(const T &&);

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

T Pop() {

return \*data[--length];

}

void Clear();

template<typename TF>

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

TStack();

private:

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

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

size\_t length, capacity;

};

#endif //OOP6\_TSTACK\_H