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 *Зинин Владислав Владимирович, 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.

Описание программы

Исходный код лежит в 14 файлах:

1. main.cpp - основная программа, взаимодействие с пользователем посредством команд из меню
2. include/figure.h - описание абстрактного класса фигур
3. include/point.h - описание класса точки

4. include/TVector.inl - реализация функций контейнера первого уровня (в моем случае вектора)

5. include/TVector.h – реализация класса контейнера первого уровня (в моем случае вектора)

6. include/rhombus.h - описание класса ромба, наследующегося от figures

7. include/point.cpp - реализация класса точки

8. include/TVectorItem.inl – реализация функций вспомогательного класса для контейнера

9. include/TVectorItem.h – описание вспомогательного класса для контейнера

10. include/rhombus.cpp: реализация класса ромба, наследующегося от figure

11. include/titerator.h – реализация класса Iterator

12. include/TQueue.h – реализация класса контейнера второго уровня

13. include/tallocation_block.h – описание класса аллокатора

14. include/ tallocation_block.cpp – реализация функций класса аллокатора

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.

Исходный код

figure.h

```
#ifndef FIGURE_H
#define FIGURE_H
#include <iostream>
#include "point.h"

class Figure
{
public:
virtual ~Figure(){};
virtual double Area() = 0;
virtual size_t VertexesNumber() = 0;
};

#endif //FIGURE_H
```

TVector.h

```
#ifndef TVECTOR_H
#define TVECTOR_H
#include <iostream>
#include "TVectorItem.h"
#include "rhombus.h"
#include <memory>

template <class T> class TVector
{
```

```
public:
/*-----init-----*/
TVector();
/*-----void-----*/
void Remove(size_t idx);
void Resize(const size_t new_size);
void InsertLast(std::shared_ptr<Rhombus> &&rhomb);
void RemoveLast();
/*-----Rhombus-----*/
const Rhombus& Last();
/*-----bool-----*/
bool Empty();
/*-----size_t-----*/
size_t Length();
/*----operator-----*/
Rhombus& operator[] (const size_t idx);
template <class B> friend std::ostream& operator<<(std::ostream& os, TVector<B> &obj);
/*-----destructor-----*/
~TVector();
private:
size_t size;
std::shared_ptr<TVectorItem<T>> first;
};

#include "TVector.inl"

#endif//TVECTOR_H
```

TVector.inl

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

/*----init----*/
template <class T> TVector<T>::TVector()
{
size = 0;
std::cout << "TVector created" << std::endl;
}
```

```
/*----bool----*/

template <class T> bool TVector<T>::Empty()
{
return size == 0?1:0;
}

template <class T> void TVector<T>::InsertLast(std::shared_ptr<Rhombus> &&rhomb)
{
std::shared_ptr<TVectorItem<T>> value (new TVectorItem<T>(rhomb));
if(size == 0)
{
this->first = value;
this->first->next = nullptr;
this->first = value;
size++;
}
else
{
std::shared_ptr<TVectorItem<T>> end = this->first;
while(end->next != nullptr)
{
end = end->next;
}
end->next = value;
value->next = nullptr;
size++;
}
}

template <class T> void TVector<T>::Resize(const size_t new_size)
{
if(size == new_size || new_size < 1)
{
return;
}
```

```
else if(new_size > size)
{
size_t iter = new_size - size;
for(int i = 0; i < iter; i++)
{
InsertLast(std::shared_ptr<Rhombus>(new Rhombus()));
}
}
else{
size_t iter = new_size;
std::shared_ptr<TVectorItem<T>> end = this->first;
for(int i = 0; i < iter; i++)
{
end = end->next;
}
end->next = nullptr;
}
size = new_size;
}

template <class T> void TVector<T>::RemoveLast()
{
if(size == 0)
{
std::cout << "List is empty" << std::endl;
}
else
{
if(size == 1)
{
size--;
std::shared_ptr<TVectorItem<T>> del = this->first;
}
else
{
```

```
std::shared_ptr<TVectorItem<T>> del = this->first;
std::shared_ptr<TVectorItem<T>> save;
while(del->next != nullptr)
{
save = del;
del = del->next;
}
size--;
save->next = nullptr;
}
}
}

template <class T> void TVector<T>::Remove(size_t idx)
{
if(idx < 1 || idx > size)
{
std::cout << "Invalid erase!" << std::endl;
}
else
{
std::shared_ptr<TVectorItem<T>> del;
std::shared_ptr<TVectorItem<T>> prev_del;
std::shared_ptr<TVectorItem<T>> next_del = this->first;
size--;
if(idx == 1)
{
del = this->first;
next_del = next_del->next;
this->first = next_del;
}
else
{
for(int i = 1; i < idx; ++i)
{
```

```
prev_del = next_del;
next_del = next_del->next;
}
del = next_del;
next_del = next_del->next;
prev_del->next = next_del;
}
}
}
/*-----Rhombus-----*/
template <class T> const Rhombus& TVector<T>::Last()
{
std::shared_ptr<TVectorItem<T>> node = this->first;
while(node->next != nullptr)
{
node = node->next;
}
return *node->rhomb;
}
/*----destructor---*/
template <class T> TVector<T>::~TVector()
{
std::cout << "TVector deleted" << std::endl;
}
/*----size_t---*/
template <class A> size_t TVector<A>::Length()
{
return size;
}
/*----operator---*/
template <class T> Rhombus& TVector<T>::operator[](const size_t idx)
{
std::shared_ptr<TVectorItem<T>> idx_rhomb = this->first;
```

```
for(int i = 1; i < idx; i++)
{
idx_rhomb = idx_rhomb->next;
}
return *idx_rhomb->rhomb;
}

template <class T> std::ostream& operator<<(std::ostream& os, TVector<T>& obj)
{
if(obj.size == 0)
{
os << "TList is empty" << std::endl;
}
else
{
os << "Print rhombus" << std::endl;
std::shared_ptr<TVectorItem<T>> print = obj.first;
os << '[';
for(int i = 0; i < obj.size - 1; i++)
{
os << print->rhomb->Area() << " " << "," << " ";
print = print->next;
}
os << print->rhomb->Area() << ']';
os << std::endl;
}
return os;
}
```

TVectorItem.h

```
#ifndef TVECTORITEM_H
#define TVECTORITEM_H
#include <iostream>
#include "rhombus.h"
#include <memory>
```

```
template <class T> class TVectorItem
{
public:
TVectorItem(std::shared_ptr<Rhombus>& rhomb);
template <class B> friend std::ostream& operator<<(std::ostream& os, TVectorItem<B> &obj);
~TVectorItem();
std::shared_ptr<T> rhomb;
std::shared_ptr<TVectorItem<T>> next;
};
#include "TVectorItem.inl"
#endif //TVECTORITEM_H
```

TVectorItem.inl

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

template <class T> TVectorItem<T>::TVectorItem(std::shared_ptr<Rhombus>& rhomb)
{
this->rhomb = rhomb;
this->next = nullptr;
}

template <class B> std::ostream& operator<<(std::ostream& os, TVectorItem<B> &obj)
{
os << obj.rhomb << " ";
return os;
}

template <class T> TVectorItem<T>::~TVectorItem()
{
std::cout << "TVectorItem deleted" << std::endl;
}
```

Main.cpp

```
#include <iostream>
#include "TVector.h"
#include <vector>
#include "tallocation_block.h"
```

```
int main()
{
TVector<Rhombus> list;
/*-----Test push_front---*/
list.InsertLast(std::shared_ptr<Rhombus>(new Rhombus(Point(1,2), Point(3,4), Point(5,6), Point(7,8))));
list.InsertLast(std::shared_ptr<Rhombus>(new Rhombus(Point(1,3), Point(3,4), Point(5,6), Point(7,8))));
list.InsertLast(std::shared_ptr<Rhombus>(new Rhombus(Point(1,4), Point(3,4), Point(5,6), Point(7,8))));
list.InsertLast(std::shared_ptr<Rhombus>(new Rhombus(Point(1,5), Point(3,4), Point(5,6), Point(7,8))));
list.InsertLast(std::shared_ptr<Rhombus>(new Rhombus(Point(1,6), Point(3,4), Point(5,6), Point(7,8))));
list.InsertLast(std::shared_ptr<Rhombus>(new Rhombus(Point(1,7), Point(3,4), Point(5,6), Point(7,8))));
std::cout << list << std::endl;
/*-----Test pop_front---*/
list.RemoveLast();
std::cout << list << std::endl;
list.RemoveLast();
std::cout << list << std::endl;
/*-----Test erase---*/
list.Resize(2);
std::cout << list << std::endl;
std::cout << "--------------" << std::endl;
std::cout << list.Length() << std::endl;
std::cout << list << std::endl;
std::cout << list[2] << std::endl;
list.Resize(4);
std::cout << list << std::endl;
list.Resize(4);
std::cout << list << std::endl;
for (auto i : list) {
std::cout << *i << std::endl;
}
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);
return 0;
}
```

Point.cpp

```
#include <iostream>
#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_;
}

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;
}

double get_x(Point &other)
{
return other.x_;
}

double get_y(Point &other)
{
return other.y_;
}

void Point::set_x(Point &other, double x)
{
other.x_ = x;
}

void Point::set_y(Point &other, double y)
{
other.y_ = y;
}
```

Point.h

```
#ifndef POINT_H
```

```
#define POINT_H

#include <iostream>

class Point
{
public:
Point();
Point(double x, double y);
Point(std::istream &is);
double dist(Point &other);
friend double get_x(Point &other);
friend double get_y(Point &other);
void set_x(Point &other, double x);
void set_y(Point &other, double y);
friend std::istream& operator>>(std::istream& is, Point& p);
friend std::ostream& operator<<(std::ostream& os, Point& p);

private:
double x_, y_;
};

#endif //POINT_H
```

Rhombus.cpp

```
#include <iostream>
#include "rhombus.h"
#include <math.h>

Rhombus::Rhombus()
{
a.set_x(a, 1);
a.set_y(a, 1);
b.set_x(b, 2);
b.set_y(b, 2);
c.set_x(c, 0);
c.set_y(c, 3);
d.set_x(d, -1);
d.set_y(d, -1);
```

```
}

Rhombus::Rhombus(std::istream &is)

{

is >> a;

is >> b;

is >> c;

is >> d;

}

Rhombus::Rhombus(Point pa, Point pb, Point pc, Point pd): a(pa), b(pb), c(pc), d(pd)

{

std::cout << "Rhombus created" << std::endl;

}

// void Rhombus::Print(std::ostream &os)

// {

// os << "Rhombus" << std::endl;

// os << a << ',' << b << ',' << c << ',' << d << std::endl;

// }

double Rhombus::Area()

{

return 0.5 * fabs(get_x(a)*get_y(b) + get_x(b)*get_y(c) + get_x(c)*get_y(d) + get_x(d)*get_y(a) - get_x(b)*get_y(a)
- get_x(c)*get_y(b) - get_x(d)*get_y(c) - get_x(a)*get_y(d));

}

Rhombus::~Rhombus()

{

std::cout << "Rhombus deleted" << std::endl;

}

size_t Rhombus::VertexesNumber()

{

return 4;

}

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

os << p.a << p.b << p.c << p.d;
```

```
return os;
}
```

Rhombus.h

```
#ifndef RHOMBUX_H
#define RHOMBUX_H
#include <iostream>
#include "point.h"
#include "figure.h"

class Rhombus : public Figure
{
public:
Rhombus();
Rhombus(std::istream &is);
Rhombus(Point a, Point b, Point c, Point d);
double Area();
size_t VertexesNumber();
friend std::ostream& operator<<(std::ostream& os, Rhombus& p);
virtual ~Rhombus();
private:
Point a, b, c, d;
};

#endif //RHOMBUX_H
```

Titerot.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->rhomb; }
```

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

void operator++() { node_ptr = node_ptr->next; }

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.h

```
#ifndef DATA_TQUEUE_H
#define DATA_TQUEUE_H

#include <iostream>

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

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

```
other.size_ = other.capacity_ = 0;
}
}

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

TQueue &operator=(TQueue &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;
}

TQueue &operator=(TQueue &&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;
}

~TQueue() {
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 TQueue<T> &vec) {
for (const T &val: vec) os << val << " ";
return os;
}

#endif
```

Tallocation_bloch.h

```
#ifndef TALLOCATION_BLOCK_H
#define TALLOCATION_BLOCK_H

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

#endif // TALLOCATION_BLOCK_H
```

Tallocation_bloch.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;
}
```