# **Лабораторная работа № 06**

**Студент**: Сазонов Вадим Кириллович , м8О-201Б-20

# **Тема: Основы работы с коллекциями: аллокаторы**

**Цель:**

* Изучение основ работы с контейнерами, знакомство концепцией аллокаторов памяти;

## **Порядок выполнения работы**

1. Ознакомиться с теоретическим материалом.

2. Получить у преподавателя вариант задания.

3. Реализовать задание своего варианта в соответствии с поставленными требованиями.

4. Подготовить тестовые наборы данных.

5. Создать репозиторий на GitHub.

6. Отправить файлы лабораторной работы в репозиторий.

7. Отчитаться по выполненной работе путём демонстрации работающей программы на тестовых наборах данных (как подготовленных самостоятельно, так и предложенных преподавателем) и ответов на вопросы преподавателя (как из числа контрольных, так и по реализации программы).

## **Требования к программе**

Разработать программу на языке C++ согласно варианту задания. Программа на C++ должна собираться с помощью системы сборки CMake. Программа должна получать данные из стандартного ввода и выводить данные в стандартный вывод.

Необходимо настроить сборку лабораторной работы с помощью CMake. Собранная программа должна называться **oop\_exercise\_06** (в случае использования Windows **oop\_exercise\_06.exe)**

Необходимо зарегистрироваться на GitHub (если студент уже имеет регистрацию на GitHub то можно использовать ее) и создать репозитарий для задания лабораторной работы.

Преподавателю необходимо предъявить ссылку на публичный репозиторий на Github. Имя репозитория должно быть<https://github.com/login/oop_exercise_06>

Где login – логин, выбранный студентом для своего репозитория на Github.

Репозиторий должен содержать файлы:

· main.cpp //файл с заданием работы

· CMakeLists.txt // файл с конфигураций CMake

· report.doc // отчет о лабораторной работе

Разработать шаблоны классов согласно варианту задания. Параметром шаблона должен являться скалярный тип данных задающий тип данных для оси координат. Классы должны иметь публичные поля. Фигуры являются фигурами вращения, т.е. равносторонние (кроме трапеции и прямоугольника). Для хранения координат фигур необходимо использовать шаблон std::pair.

Вариант 4:

|  |  |  |  |
| --- | --- | --- | --- |
| 4. | Трапеция | Стек | Очередь |

Код программы:

Main.cpp

#include <iostream>

#include "trapezoid.h"

#include "queue.h"

#include "stack.h"

void TestTStack()

{

TStack<Trapezoid> stack;

Trapezoid trap(1,2,3,4);

stack.Push(trap);

while (!stack.IsEmpty())

{

std::cout << stack.Top() << std::endl;

stack.Pop();

}

}

void TestAllocationBlock()

{

TAllocationBlock allocator(sizeof(int), 3);

int \*a1 = nullptr;

int \*a2 = nullptr;

int \*a3 = nullptr;

a1 = (int \*)allocator.allocate();

\*a1 = 1;

std::cout << "a1 pointer value:" << \*a1 << std::endl;

a2 = (int \*)allocator.allocate();

\*a2 = 2;

a3 = (int \*)allocator.allocate();

\*a3 = 3;

std::cout << "a2 pointer value:" << \*a2 << std::endl;

allocator.deallocate(a1);

allocator.deallocate(a2);

allocator.deallocate(a3);

}

int main()

{

TestAllocationBlock();

TestTStack();

return 0;

}

Allocationblock.h

#ifndef TALLOCATION\_BLOCK\_H

#define TALLOCATION\_BLOCK\_H

#include <cstdlib>

#include "queue.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\*> \_free\_blocks;

size\_t \_free\_count;

};

#endif

Allocationblock.cpp

#include "allocationblock.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++) {

\_free\_blocks.Push(\_used\_blocks + i \* size);

}

\_free\_count = count;

std::cout << "Memory init" << "\n";

}

void\* TAllocationBlock::allocate()

{

void\* result = nullptr;

if (\_free\_count == 0) {

std::cout << "No memory exception\n" << "\n";

return result;

}

result = \_free\_blocks.Top();

\_free\_blocks.Pop();

--\_free\_count;

std::cout << "Allocate " << (\_count - \_free\_count) << "\n";

return result;

}

void TAllocationBlock::deallocate(void\* pointer)

{

\_free\_blocks.Push(pointer);

++\_free\_count;

std::cout << "Deallocated block\n";

}

bool TAllocationBlock::has\_free\_blocks()

{

return \_free\_count > 0;

}

TAllocationBlock::~TAllocationBlock()

{

free(\_used\_blocks);

}

Iter.h

#ifndef LAB5\_TITERATOR\_H

#define LAB5\_TITERATOR\_H

#include <iostream>

template <class item, class T> class TIterator {

public:

TIterator(std::shared\_ptr<item> n) {

item\_ptr = n;

}

std::shared\_ptr<T> operator \*() {

return item\_ptr->polygon;

}

std::shared\_ptr<T> operator ->() {

return item\_ptr->polygon;

}

void operator ++() {

item\_ptr = item\_ptr->next;

}

TIterator operator ++(int) {

TIterator iter(\*this);

++(\*this);

return iter;

}

bool operator ==(TIterator const& i) {

return (item\_ptr == i.item\_ptr);

}

bool operator !=(TIterator const& i) {

return (item\_ptr != i.item\_ptr);

}

private:

std::shared\_ptr<item> item\_ptr;

};

#endif //LAB5\_TITERATOR\_H

Queue.cpp

#include <iostream>

#include <memory>

#include "queue.h"

#include "queueitem.h"

template <class T>

TQueue<T>::TQueue()

{

head = nullptr;

count = 0;

}

template <class T>

void TQueue<T>::Push(const T &item)

{

TQueueItem<T> \*tmp = new TQueueItem<T>(item, head);

head = tmp;

++count;

}

template <class T>

bool TQueue<T>::IsEmpty() const

{

return !count;

}

template <class T>

uint32\_t TQueue<T>::GetSize() const

{

return count;

}

template <class T>

void TQueue<T>::Pop()

{

if(head) {

TQueueItem<T> \*tmp = &head->GetNext();

delete head;

head = tmp;

--count;

}

}

template <class T>

T &TQueue<T>::Top()

{

return head->Pop();

}

template <class T>

TQueue<T>::~TQueue()

{

for(TQueueItem<T> \*tmp = head, \*tmp2; tmp; tmp = tmp2) {

tmp2 = &tmp->GetNext();

delete tmp;

}

}

template class

TQueue<void \*>;

Queue.h

#ifndef TQUEUE\_H

#define TQUEUE\_H

#include <iostream>

#include <memory>

#include "queueitem.h"

template <class T>

class TQueue

{

public:

TQueue();

virtual ~TQueue();

void Push(const T &item);

void Pop();

T &Top();

bool IsEmpty() const;

uint32\_t GetSize() const;

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

private:

TQueueItem<T> \*head;

uint32\_t count;

};

#endif

Queueitem.cpp

#include <iostream>

#include <memory>

#include "queueitem.h"

template <class T>

TQueueItem<T>::TQueueItem(const T &val, TQueueItem<T> \*item)

{

value = new T(val);

next = item;

}

template <class T>

void TQueueItem<T>::Push(const T &val)

{

\*value = val;

}

template <class T>

T &TQueueItem<T>::Pop() const

{

return \*value;

}

template <class T>

void TQueueItem<T>::SetNext(TQueueItem<T> \*item)

{

next = item;

}

template <class T>

TQueueItem<T> &TQueueItem<T>::GetNext() const

{

return \*next;

}

template <class T>

TQueueItem<T>::~TQueueItem()

{

delete value;

}

template class

TQueueItem<void \*>;

Queueitem.h

#ifndef TSTACKITEM\_H

#define TSTACKITEM\_H

#include <iostream>

#include <memory>

template <class T>

class TQueueItem

{

public:

TQueueItem(const T &val, TQueueItem<T> \*item);

virtual ~TQueueItem();

void Push(const T &val);

T &Pop() const;

void SetNext(TQueueItem<T> \*item);

TQueueItem<T> &GetNext() const;

private:

T \*value;

TQueueItem<T> \*next;

};

#endif

Stack.cpp

#include <iostream>

#include <memory>

#include "stack.h"

#include "trapezoid.h"

template <class T>

TStack<T>::TStack()

{

head = nullptr;

count = 0;

}

template <class T>

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

{

TStackItem<T> \*tmp = new TStackItem<T>(item, head);

head = tmp;

++count;

}

template <class T>

bool TStack<T>::IsEmpty() const

{

return !count;

}

template <class T>

uint32\_t TStack<T>::GetSize() const

{

return count;

}

template <class T>

void TStack<T>::Pop()

{

if(head) {

TStackItem<T> \*tmp = &head->GetNext();

delete head;

head = tmp;

--count;

}

}

template <class T>

T &TStack<T>::Top()

{

return head->Pop();

}

template <class T>

TStack<T>::~TStack()

{

for(TStackItem<T> \*tmp = head, \*tmp2; tmp; tmp = tmp2) {

tmp2 = &tmp->GetNext();

delete tmp;

}

}

typedef unsigned char Byte;

template class

TStack<Trapezoid>;

Stack.h

#pragma once

#ifndef OOP\_LAB4\_TSTACK\_H

#define OOP\_LAB4\_TSTACK\_H

#include <iostream>

#include <memory>

#include "stackitem.h"

template <class T>

class TStack

{

public:

TStack();

virtual ~TStack();

void Push(const T &item);

void Pop();

T &Top();

bool IsEmpty() const;

uint32\_t GetSize() const;

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

private:

TStackItem<T> \*head;

uint32\_t count;

};

#endif //OOP\_LAB4\_TSTACK\_H

Stackitem.cpp

#include <iostream>

#include <memory>

#include "stackitem.h"

template <class T>

TStackItem<T>::TStackItem(const T &val, TStackItem<T> \*item)

{

value = new T(val);

next = item;

}

template <class T>

void TStackItem<T>::Push(const T &val)

{

\*value = val;

}

template <class T>

T &TStackItem<T>::Pop() const

{

return \*value;

}

template <class T>

void TStackItem<T>::SetNext(TStackItem<T> \*item)

{

next = item;

}

template <class T>

TStackItem<T> &TStackItem<T>::GetNext() const

{

return \*next;

}

template <class T>

TStackItem<T>::~TStackItem()

{

delete value;

}

// typedef unsigned char Byte;

#include "trapezoid.h"

template class

TStackItem<Trapezoid>;

Stackitem.h

#pragma once

#ifndef OOP\_LAB4\_TSTACKITEM\_H

#define OOP\_LAB4\_TSTACKITEM\_H

#include <iostream>

#include <memory>

template <class T>

class TStackItem

{

public:

TStackItem(const T &val, TStackItem<T> \*item);

virtual ~TStackItem();

void Push(const T &val);

T &Pop() const;

void SetNext(TStackItem<T> \*item);

TStackItem<T> &GetNext() const;

private:

T \*value;

TStackItem<T> \*next;

};

#endif //OOP\_LAB4\_TSTACKITEM\_H

Trapezoid.cpp

#include "trapezoid.h"

#include <cmath>

Trapezoid::Trapezoid()

: len\_ab(0.0),

len\_bc(0.0),

len\_cd(0.0),

len\_da(0.0) {

}

Trapezoid::Trapezoid(double ab, double bc, double cd, double da)

: len\_ab(ab),

len\_bc(bc),

len\_cd(cd),

len\_da(da) {

}

Trapezoid::Trapezoid(std::shared\_ptr<Trapezoid>& other)

: Trapezoid(other->len\_ab, other->len\_bc, other->len\_cd, other->len\_da) {

}

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

if (this == &other)

return \*this;

len\_ab = other.len\_ab;

len\_bc = other.len\_bc;

len\_cd = other.len\_cd;

len\_da = other.len\_da;

std::cout << "Trapezoid copied" << std::endl;

return \*this;

}

std::ostream& operator<<(std::ostream& os, const Trapezoid& obj){

os << obj.len\_ab <<" " << obj.len\_bc << " " << obj.len\_cd << " " << obj.len\_da;

return os;

}

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

if (this->len\_ab == other.len\_ab &&

this->len\_bc == other.len\_bc &&

this->len\_cd == other.len\_cd &&

this->len\_da == other.len\_da) {

std::cout << "Trapezoids are equal" << std::endl;

return 1;

} else {

std::cout << "Trapezoids are not equal" << std::endl;

return 0;

}

}

size\_t Trapezoid::VertexesNumber() {

return 4;

}

double Trapezoid::Area() {

double p = (len\_ab + len\_bc + len\_cd + len\_da) / 2;

return (len\_bc + len\_da) \*

std::sqrt((p - len\_bc) \*

(p - len\_da) \*

(p - len\_da - len\_ab) \*

(p - len\_da - len\_cd)) /

std::abs(len\_bc - len\_da);

}

Trapezoid::~Trapezoid() {

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

}

TAllocationBlock Trapezoid::block(sizeof(Trapezoid), 1000);

void \*Trapezoid::operator new(size\_t size) {

return block.allocate();

}

void Trapezoid::operator delete(void \*ptr) {

block.deallocate(ptr);

}

Trapezoid.h

#ifndef TRAPEZOID\_H

#define TRAPEZOID\_H

#include <iostream>

#include <memory>

#include "allocationblock.h"

class Trapezoid {

public:

Trapezoid();

Trapezoid(double a, double b, double c, double d);

Trapezoid(std::shared\_ptr<Trapezoid>& other);

friend std::istream& operator>>(std::istream& is, Trapezoid& obj);

friend std::ostream& operator<<(std::ostream& os, const Trapezoid& obj);

Trapezoid& operator=(const Trapezoid& right);

bool operator==(const Trapezoid& right);

virtual ~Trapezoid();

size\_t VertexesNumber();

double Area();

void \*operator new(size\_t size);

void operator delete(void \*ptr);

public:

double len\_ab, len\_bc, len\_cd, len\_da;

private:

static TAllocationBlock block;

};

#endif // TRAPEZOID\_H