

# Лабораторная работа №6

## 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.

Алокатор 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.

Листинг

**Fi  
g  
ur  
e.  
h**

//	
	// Created by Илья Рожков on 12.09.2021.
//	
	#ifndef LAB1_FIGURE_H

	#define LAB1_FIGURE_H
	#include "iostream"
	#include <utility>
	#include <math.h>
	#include <cmath>
	class Figure {
	public:
	virtual void Print() const = 0;
	virtual size_t VertexesNumber() const = 0;
	virtual double Area() const = 0;
	};
	#endif //LAB1_FIGURE_H

**G  
er  
o  
n  
F  
or  
m  
ul  
a.**

**c  
p  
p**

//

// Created by Илья Рожков on 16.09.2021.

//

#include "GeronFormula.h"

#include <cmath>

double GeronFormula(double a, double b, double c) {

double p, s;

p = (a + b + c) / 2;

s = sqrt(p \* (p - a) \* (p - b) \* (p - c));

return s;

}

double getDistance(const std::pair<double, double> &x, const std::pair<double,

return sqrt(pow((x.first - y.first), 2) + pow((x.second - y.second), 2));

}

double GeronFormulaFromCoordinates(const Coordinate &a, const Coordinate &b,

double x = getDistance(a, b);

double y = getDistance(b, c);

double z = getDistance(c, a);

return GeronFormula(x, y, z);

}

double AreaOfMultigone(const std::vector<Coordinate> &coordinates) {

double s = 0;

for (int i = 0; i < coordinates.size(); i += 3)

	s += GeronFormulaFromCordinates(cordinates[i], cordinates[(i + 1) %
	return s;
	}

# G e r o n F o r m u l a .h

//
----

// Created by Илья Рожков on 16.09.2021.
--

//
----

#ifndef LAB1_GERONFORMULA_H
-----------------------------

#define LAB1_GERONFORMULA_H
-----------------------------

#include <utility>
--------------------

#include <vector>
-------------------

typedef std::pair<double, double> Cordinate;
--

double GeronFormula(double a, double b, double c);
--

double getDistance(const std::pair<double, double>& x , const std::pair<double,
---

double GeronFormulaFromCordinates(const Cordinate& a, const Cordinate&
--

double AreaOfMultigone(const std::vector<Cordinate>& cordinates);
---

	#endif //LAB1_GERONFORMULA_H
//	
	// Created by Илья Рожков on 16.09.2021.
	//
	#include "Hexagon.h"
	Hexagon::Hexagon() {
	for (int i = 0; i < 6; i++) {
	Coordinate elemt = std::make_pair(0, 0);
	_coordinates.push_back(elemt);
	}
	}
	Hexagon::Hexagon(const std::vector<Coordinate> &coordinates) :
	if (_coordinates.size() != 6) {
	throw "wrong size";
	}
	}
	size_t Hexagon::VertexesNumber() const {
	return 6;
	}
	double Hexagon::Area() const {
	return AreaOfMultigone(_coordinates);
	}

	void Hexagon::Print() const {
	for (int i = 0; i < _coordinates.size(); i++)
	std::cout << _coordinates[i].first << ' ' << _coordinates[i].second << std::endl;
	}
	std::ostream &operator<<(std::ostream &out, const Hexagon &r) {
	for (int i = 0; i < r._coordinates.size(); i++)
	out << r._coordinates[i].first << ' ' << r._coordinates[i].second << std::endl;
	return out;
	}
	std::istream &operator>>(std::istream &in, Hexagon &r) {
	for (int i = 0; i < 6; i++)
	in >> r._coordinates[i].first >> r._coordinates[i].second;
	return in;
	}
	Hexagon::~~Hexagon() {
	}

**H  
ex  
a  
g  
o  
n.  
h**

//	
	// Created by Илья Рожков on 16.09.2021.
	//

```

#ifndef LAB1_HEXAGON_H
#define LAB1_HEXAGON_H

#include "Figure.h"
#include "GeronFormula.h"


class Hexagon : public Figure {
public:
    Hexagon();
    ~Hexagon();
    Hexagon(const std::vector<Cordinate>& cordinates);
    size_t VertexesNumber() const override;
    double Area() const override;
    void Print() const override;
    friend std::ostream& operator<<(std::ostream &out, const Hexagon& r);
    friend std::istream& operator>> (std::istream &in, Hexagon& r);


protected:
    std::vector<Cordinate> _cordinates;

};

#endif //LAB1_HEXAGON_H

```

**tb  
in  
ar  
yt  
re  
e.  
c  
p  
p**



```

{
    if (curr->get_poligon() == octagon)
    {
        curr->set_count_fig(curr->get_count_fig() + 1);
        return;
    }
    if (octagon.Area() < curr->get_poligon().Area())
    if (curr->get_left() == nullptr)
    {
        curr->set_left(OctSptr);
        return;
    }
    if (octagon.Area() >= curr->get_poligon().Area())
    if (curr->get_right() == nullptr && !(curr->get_poligon() == octagon))
    {
        curr->set_right(OctSptr);
        return;
    }
    if (curr->get_poligon().Area() > octagon.Area())
    curr = curr->get_left();
    else
    curr = curr->get_right();
}
}

template <class Poligon>
const Poligon& TBinaryTree<Poligon>::GetItemNotLess(double area) {
    SPTR(TreeElem<Poligon>) curr = t_root;
    while (curr)
    {

```

```
if (area == curr->get_poligon().Area())
```

```
return curr->get_poligon();
```

```
if (area < curr->get_poligon().Area())
```

```
{
```

```
curr = curr->get_left();
```

```
continue;
```

```
}
```

```
if (area >= curr->get_poligon().Area())
```

```
{
```

```
curr = curr->get_right();
```

```
continue;
```

```
}
```

```
}
```

```
throw std::out_of_range("out of range");
```

```
}
```

```
template <class Poligon>
```

```
size_t TBinaryTree<Poligon>::Count(const Poligon& octagon) {
```

```
size_t count = 0;
```

```
SPTR(TreeElem<Poligon>) curr = t_root;
```

```
while (curr)
```

```
{
```

```
if (curr->get_poligon() == octagon)
```

```
count = curr->get_count_fig();
```

```
if (octagon.Area() < curr->get_poligon().Area())
```

```
{
```

```
curr = curr->get_left();
```

```
continue;
```

```
}
```

```
if (octagon.Area() >= curr->get_poligon().Area())
```

```
{
```

```
curr = curr->get_right();
```

```
continue;
```

```
}
```

```
}
```

```
return count;
```

```
}
```

```
template <class Poligon>
```

```
void Pop_List(SPTR(TreeElem<Poligon>) curr, SPTR(TreeElem<Poligon>)
```

```
if (parent->get_left() == curr)
```

```
parent->set_left(nullptr);
```

```
else
```

```
parent->set_right(nullptr);
```

```
}
```

```
template <class Poligon>
```

```
void Pop_Part_of_Branch(SPTR(TreeElem<Poligon>) curr,
```

```
if (parent) {
```

```
if (curr->get_left()) {
```

```
if (parent->get_left() == curr)
```

```
parent->set_left(curr->get_left());
```

```
if (parent->get_right() == curr)
```

```
parent->set_right(curr->get_left());
```

```
curr->set_right(nullptr);
```

```
curr->set_left(nullptr);
```

```
return;
```

```
}
```

```

if (curr->get_left() == nullptr) {
    if (parent && parent->get_left() == curr)
        parent->set_left(curr->get_right());

    if (parent && parent->get_right() == curr)
        parent->set_right(curr->get_right());

    curr->set_right(nullptr);
    curr->set_left(nullptr);
    return;
}

}

template <class Poligon>
void Pop_Root_of_Subtree(SPTR(TreeElem<Poligon>) curr,
    SPTR(TreeElem<Poligon>) replace = curr->get_left();
    SPTR(TreeElem<Poligon>) rep_parent = curr;
    while (replace->get_right())
    {
        rep_parent = replace;
        replace = replace->get_right();
    }

    curr->set_poligon(replace->get_poligon());
    curr->set_count_fig(replace->get_count_fig());

    if (rep_parent->get_left() == replace)
        rep_parent->set_left(nullptr);
    else

```

```
rep_parent->set_right(nullptr);
```

```
return;
```

```
}
```

```
template <class Poligon>
```

```
void TBinaryTree<Poligon>::Pop(const Poligon& octagon) {
```

```
SPTR(TreeElem<Poligon>) curr = t_root;
```

```
SPTR(TreeElem<Poligon>) parent = nullptr;
```

```
while (curr && curr->get_poligon() != octagon)
```

```
{
```

```
parent = curr;
```

```
if (curr->get_poligon().Area() > octagon.Area())
```

```
curr = curr->get_left();
```

```
else
```

```
curr = curr->get_right();
```

```
}
```

```
if (curr == nullptr)
```

```
return;
```

```
curr->set_count_fig(curr->get_count_fig() - 1);
```

```
if(curr->get_count_fig() <= 0)
```

```
{
```

```
if (curr->get_left() == nullptr && curr->get_right() == nullptr)
```

```
{
```

```
Pop_List(curr, parent);
```

```
return;
```

```
}
```

```
if (curr->get_left() == nullptr || curr->get_right() == nullptr)
```

```
{
```

```
Pop_Part_of_Branch(curr, parent);
```

```
return;
```

```
}
```

```
if (curr->get_left() != nullptr && curr->get_right() != nullptr)
```

```
{
```

```
Pop_Root_of_Subtree(curr, parent);
```

```
return;
```

```
}
```

```
}
```

```
}
```

```
template <class Poligon>
```

```
bool TBinaryTree<Poligon>::Empty() {
```

```
return t_root == nullptr ? true : false;
```

```
}
```

```
template <class Poligon>
```

```
void Tree_out (std::ostream& os, SPTR(TreeElem<Poligon>) curr) {
```

```
if (curr)
```

```
{
```

```
if(curr->get_poligon().Area() >= 0)
```

```
os << curr->get_count_fig() << "*" << curr->get_poligon().Area();
```

```
if(curr->get_left() || curr->get_right())
```

```
{
```

```
os << ": [";
```

```
if (curr->get_left())
```

```
Tree_out(os, curr->get_left());
```

```
if(curr->get_left() && curr->get_right())
```

```
os << " , ";
```

```
if (curr->get_right())
```

```
Tree_out(os, curr->get_right());
```

```
os << "]" ;
```

```
}
```

```
}
```

```
}
```

```
template <class A>
```

```
std::ostream& operator<<(std::ostream& os, const TBinaryTree<A>& tree) {
```

```
SPTR(TreeElem<A>) curr = tree.t_root;
```

```
Tree_out(os, curr);
```

```
return os;
```

```
}
```

```
template <class Poligon>
```

```
void recursive_clear(SPTR(TreeElem<Poligon>) curr){
```

```
if(curr)
```

```
{
```

```
if (curr->get_left())
```

```
recursive_clear(curr->get_left());
```

```
curr->set_left(nullptr);
```

```
if (curr->get_right())
```

```
recursive_clear(curr->get_right());
```

```
curr->set_right(nullptr);
```

```
}
```

```
}
```

```
template <class Poligon>
```

```
void TBinaryTree<Poligon>::Clear() {
```

```
if (t_root->get_left())
```

```
recursive_clear(t_root->get_left());
```

```

t_root->set_left(nullptr);
if (t_root->get_right())
recursive_clear(t_root->get_right());
t_root->set_right(nullptr);
t_root = nullptr;
}

template <class Poligon>
TBinaryTree<Poligon>::~TBinaryTree() {
}

#include "pentagon.h"
template class TBinaryTree<Pentagon>;
template std::ostream& operator<<(std::ostream& os, const

```

## Node.cpp

```

//
// Created by Илья Рожков on 30.09.2021.
//

#include "Node.h"
#include <memory>

template <class Poligon>

```



	TreeElem<Poligon>::TreeElem() {
	polig = nullptr;
	count_fig = 0;
	t_left = nullptr;
	t_right = nullptr;
	}
	template <class Poligon>
	TreeElem<Poligon>::TreeElem(const Poligon poligon) {
	polig = MakeSPTR(Poligon)(poligon);
	count_fig = 1;
	t_left = nullptr;
	t_right = nullptr;
	}
	template <class Poligon>
	const Poligon& TreeElem<Poligon>::get_poligon() const{
	return *polig;
	}
	template <class Poligon>
	int TreeElem<Poligon>::get_count_fig() const{
	return count_fig;
	}
	template <class Poligon>
	SPTR(TreeElem<Poligon>) TreeElem<Poligon>::get_left() const{
	return t_left;
	}
	template <class Poligon>
	SPTR(TreeElem<Poligon>) TreeElem<Poligon>::get_right() const{
	return t_right;

	}
	template <class Poligon>
	void TreeElem<Poligon>::set_poligon(const Poligon& poligon){
	polig = MakeSPTR(Poligon)(poligon);
	}
	template <class Poligon>
	void TreeElem<Poligon>::set_count_fig(const int count) {
	count_fig = count;
	}
	template <class Poligon>
	void TreeElem<Poligon>::set_left(SPTR(TreeElem<Poligon>) to_left) {
	t_left = to_left;
	}
	template <class Poligon>
	void TreeElem<Poligon>::set_right(SPTR(TreeElem<Poligon>) to_right) {
	t_right = to_right;
	}
	template <class Poligon>
	TreeElem<Poligon>::~~TreeElem() {
	}
	#include "Pentagon.h"
	template class TreeElem<Pentagon>;



#ifndef ITERATOR_H	
	#define ITERATOR_H
	#include <iostream>
	#include <memory>
	template <class Poligon>
	class Iterator {
	public:
	Iterator(std::shared_ptr<Poligon>* n){
	iter = n;
	}
	Poligon operator*(){
	return *(*iter);
	}
	Poligon operator->(){
	return *(*iter);
	}
	void operator++(){
	iter += 1;
	}
	Iterator operator++(int){
	Iterator iter(*this);
	++(*this);
	return iter;
	}
	bool operator==(Iterator const& i) const{
	return iter == i.iter;

	}
	bool operator!=(Iterator const& i) const{
	return iter != i.iter;
	}
	private:
	std::shared_ptr<Poligon>* iter;
	};
	#endif

## tvector.hpp

#ifndef	
	#define TVECTOR_H
	#include <iostream>
	#include "iterator.hpp"
	#include <memory>
	#define SPTR(T) std::shared_ptr<T>
	template <class Polygon>
	class TVector
	{
	public:
	// Конструктор по умолчанию
	TVector();
	// изменение размера массива
	void Resize(size_t nsize);
	// Конструктор копирования

	TVector(const TVector& other);
	// Метод, добавляющий фигуру в конец массива
	void InsertLast(const Polygon& polygon);
	// Метод, удаляющий последнюю фигуру массива
	void RemoveLast();
	// Метод, возвращающий последнюю фигуру массива
	const Polygon& Last();
	// Перегруженный оператор обращения к массиву по индексу
	const SPTR(Polygon) operator[] (const size_t idx);
	// Метод, проверяющий пустоту
	bool Empty();
	// Метод, возвращающий длину массива
	size_t Length();
	// Оператор вывода для массива в формате:
	// "[S1 S2 ... Sn]", где Si - площадь фигуры
	template <class T>
	friend std::ostream& operator<<(std::ostream& os, const
	// Метод, удаляющий все элементы контейнера,
	// но позволяющий пользоваться им.
	void Clear();
	// Итератор начала
	Iterator<Polygon> begin(){
	return Iterator<Polygon>(data);
	}
	// Итератор конца
	Iterator<Polygon> end(){
	return Iterator<Polygon>(data + size);
	}
	// Деструктор
	virtual ~TVector();

	private:
	int size;
	SPTR(Polygon)* data;
	};
	#endif
	template <class Polygon>
	TVector<Polygon>::TVector(){
	size = 1;
	data = new SPTR(Polygon)[size];
	}
	template <class Polygon>
	void TVector<Polygon>::Resize(size_t nsize){
	if(nsize == size)
	return;
	else{
	SPTR(Polygon)* ndata = new SPTR(Polygon)[nsize];
	for (int i = 0; i < (size < nsize ? size : nsize); i++)
	ndata[i] = data[i];
	delete[] data;
	data = ndata;
	size = nsize;
	}
	}
	template <class Polygon>
	TVector<Polygon>::TVector(const TVector& other){

	size = other.size;
	data = new SPTR(Polygon)[other.size];
	for (int i = 0; i < size; i++)
	data[i] = other.data[i];
	}
	template <class Polygon>
	void TVector<Polygon>::InsertLast(const Polygon& polygon){
	if (data[size - 1] != nullptr)
	Resize(size+1);
	data[size - 1] = std::make_shared<Polygon>(polygon);
	}
	template <class Polygon>
	void TVector<Polygon>::RemoveLast(){
	data[size-1]=nullptr;
	}
	template <class Polygon>
	const Polygon& TVector<Polygon>::Last(){
	return *(data[size - 1]);
	}
	template <class Polygon>
	const SPTR(Polygon) TVector<Polygon>::operator[] (const size_t idx)
	if (idx >= 0 && idx < size)
	return data[idx];
	exit(1);
	}

	template <class Polygon>	
	bool TVector<Polygon>::Empty(){	
	return size == 0;	
	}	
	template <class Polygon>	
	size_t TVector<Polygon>::Length(){	
	return size;	
	}	
	template <class Polygon>	
	std::ostream& operator<<(std::ostream& os, const	
	os << '[';	
	for (size_t i = 0; i < arr.size; i++)	
	os << (arr.data[i]->Area() << ((i != arr.size-1) ? ' ' : '\0');	
	os << ']';	
	return os;	
	}	
	template <class Polygon>	
	void TVector<Polygon>::Clear(){	
	delete[] data;	
	size = 1;	
	data = new SPTR(Polygon)[size];	
	}	
	template <class Polygon>	
	TVector<Polygon>::~TVector(){	
	delete[] data;	
	}	



LinkedListItem.h

#ifndef ITEM2_H	
	#define ITEM2_H
	#include <memory>
	class Item2 {
	public:
	Item2(void *ptr);
	Item2* to_right(Item2* next);
	Item2* Next();
	void* GetItem();
	virtual ~Item2();
	private:
	void* link;
	Item2* next;
	};

	#endif // ITEM2_H
	Item2::Item2(void* link) {
	this->link = link;
	this->next = nullptr;
	}
	Item2* Item2::to_right(Item2* next) {
	Item2* set = this->next;
	this->next = next;
	return set;
	}
	Item2* Item2::Next() {
	return this->next;
	}
	void* Item2::GetItem() {
	return this->link;
	}
	Item2::~~Item2() {}