**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №5**

**по дисциплине «Объектно-ориентированное программирование»**

Тема: «Полиморфная логика»

|  |  |  |
| --- | --- | --- |
| Студент гр. 6381 |  | Мейзер Д.В. |
| Преподаватель |  | Терентьев А.О. |

Санкт-Петербург

2018

**Цель работы.**

Объединение предыдущих лабораторных работ в приложении, использующем логику полиморфного хранения объектов.

**Задание 1**. Немодифицирующий алгоритм.

1. Поиск первого элемента, удовлетворяющего определенному критерию.

Решение:

Функция, принимающая на вход контейнер и предикат, и возвращающая указатель на первый элемент, удовлетворяющий предикату:

stepik::shared\_ptr<Shape> findFirstOccurence(

const stepik::vector<stepik::shared\_ptr<Shape>>& SSPVect,

bool (\*predicate)(const Shape&))

{

for (const stepik::shared\_ptr<Shape> &i : SSPVect)

{

if (predicate(\*i))

return i;

}

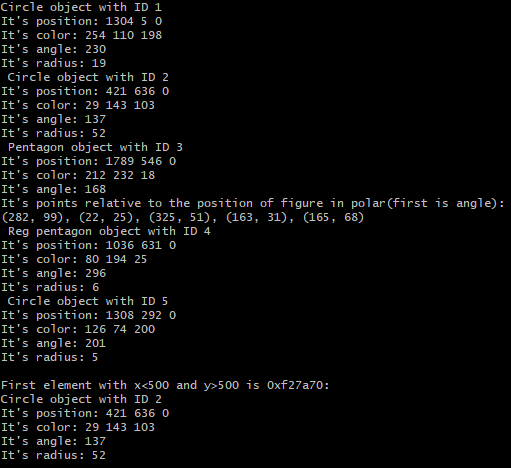
return stepik::shared\_ptr<Shape>{nullptr};

}

**Тестирование.**

В качестве предиката используется проверка того, что у фигуры x<500 и y>500:

Рис. 1.



**Задание 2**.

12. Отсортировать элементы диапазона так, чтобы правильный элемент находился на n-й позиции (все стоящие перед ним элементы не превосходят его, а все стоящие после - превосходят либо равны).

Решение:

Функция, сортирующая элементы так, что в начале диапазона [begin, end] контейнера находятся элементы, которые не удовлетворяют предикату, и возвращающая умный указатель на первый элемент, удовлетворяющий предикату, или пустой умный указатель в ином случае:

stepik::shared\_ptr<Shape> sort(stepik::vector<stepik::shared\_ptr<Shape>>& SSPVect,

unsigned begin, unsigned end, bool (\*predicate)(const Shape&))

{

if ((begin > SSPVect.size()) || (end > SSPVect.size()) || (end < begin))

throw out\_of\_range("Bad arguments in sort function");

// Sorting

std::sort(SSPVect.begin()+begin, SSPVect.begin()+end,

[predicate](const stepik::shared\_ptr<Shape>& lhs,

const stepik::shared\_ptr<Shape>& rhs) -> bool

{

return predicate(\*lhs)<predicate(\*rhs);

});

// Searching first matching element and returning it's index if found

for (unsigned i = begin; i < end; ++i)

if (predicate(\*SSPVect[i]))

return SSPVect[i];

// No matching element

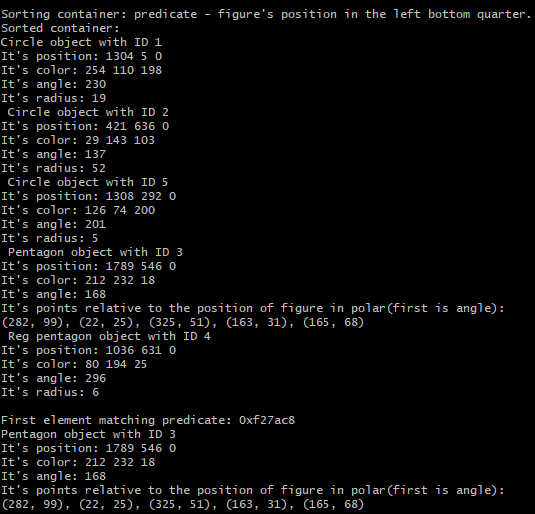
return stepik::shared\_ptr<Shape>{nullptr};

}

**Тестирование.**

Для того же контейнера, что и в тестировании для задания 1:

Рис. 2.



**Вывод.**

В процессе выполнения данной лабораторной работы были объединены предыдущие лабораторные работы в приложении, использующем логику полиморфного хранения объектов и выполняющем немодифицирующий (нахождение первого элемента контейнера, соответствующего предикату) и модифицирующий (сортировка элементов контейнера, такая, что в начале находятся элементы, не удовлетворяющие предикату), алгоритмы на контейнере объектов.

**Приложение. Исходный код программы.**

**main.cpp**

#include "Figures.hpp"

#include "vector.cpp"

#include "shared\_ptr.cpp"

#include <iostream>

#include <ctime>

#define OBJ\_N 5 // Количество тестируемых объектов

#define X\_MAX 1920 // Максимально возможное значение X положения фигуры

#define Y\_MAX 1080 // Y

#define R\_MAX 100 // Максимальный радиус от цента фигуры до ее вершин

using namespace std;

void generateShapes(stepik::vector<stepik::shared\_ptr<Shape>>& SSPVect)

{

srand(time(NULL));

for (stepik::shared\_ptr<Shape> &i : SSPVect)

{

switch (rand()%3)

{

case 0: // Круг

i.reset(new Circle{});

static\_cast<Circle&>(\*i).setRadius(

static\_cast<double>(rand()%R\_MAX));

break;

case 1: // Пятиугольник

i.reset(new Pentagon{});

static\_cast<Pentagon&>(\*i).setDotsRadius({

static\_cast<double>(rand()%R\_MAX),

static\_cast<double>(rand()%R\_MAX),

static\_cast<double>(rand()%R\_MAX),

static\_cast<double>(rand()%R\_MAX),

static\_cast<double>(rand()%R\_MAX)

});

static\_cast<Pentagon&>(\*i).setDotsAngle({

static\_cast<double>(rand()),

static\_cast<double>(rand()),

static\_cast<double>(rand()),

static\_cast<double>(rand()),

static\_cast<double>(rand())

});

break;

case 2: // Правильный пятиугольник

i.reset(new RegPentagon{});

static\_cast<RegPentagon&>(\*i).setDotsRadius(

static\_cast<double>(rand()%R\_MAX));

break;

}

// Устанавливаем поля базового для всех объектов класса

i->setPos({static\_cast<double>(rand()%X\_MAX),

static\_cast<double>(rand()%Y\_MAX), 0});

i->setColor({rand()%256, rand()%256, rand()%256});

i->setAngle(rand());

}

}

// First argument is container with shapes, second - predicate function

// Returns a shared pointer to a found element

// Or empty shared pointer if all elements not matching a predicate

stepik::shared\_ptr<Shape> findFirstOccurence(

const stepik::vector<stepik::shared\_ptr<Shape>>& SSPVect,

bool (\*predicate)(const Shape&))

{

for (const stepik::shared\_ptr<Shape> &i : SSPVect)

{

if (predicate(\*i))

return i;

}

return stepik::shared\_ptr<Shape>{nullptr};

}

// Function that sorts container(First elements are not matching predicate)

// First argument is container with shapes, second - predicate function

// Returns first element that matching a predicate.

// If there is no such elements then returns -1

int sort(stepik::vector<stepik::shared\_ptr<Shape>>& SSPVect,

bool (\*predicate)(const Shape&))

{

// Sorting

std::sort(SSPVect.begin(), SSPVect.end(),

[predicate](const stepik::shared\_ptr<Shape>& lhs,

const stepik::shared\_ptr<Shape>& rhs) -> bool

{

return predicate(\*lhs)<predicate(\*rhs);

});

// Searching first matching element and returning it's index if found

for (unsigned i = 0, size = SSPVect.size(); i < size; ++i)

if (predicate(\*SSPVect[i]))

return i;

// No matching element

return -1;

}

int main()

{

stepik::vector<stepik::shared\_ptr<Shape>> SSPVect(OBJ\_N);

generateShapes(SSPVect);

for (auto &i : SSPVect)

{

cout << \*i << " ";

}

cout << endl;

//1. Поиск первого элемента удовлетворяющего определенному критерию.

{

stepik::shared\_ptr<Shape> found{findFirstOccurence(SSPVect,

[](const Shape& a)

{

if (a.getPos().x<500 && a.getPos().y>500) return true;

return false;

})};

if (found)

cout << "First element with x<500 and y>500 is " << found.get() << ": "

<< endl << \*found;

else

{

cout << "Couldn't find element with x<500 and y>500." << endl;

}

}

cout << endl;

// 12. Отсортировать элементы диапазона так, чтобы правильный элемент находился на n-й позиции

// (все стоящие перед ним элементы не превосходят его, а все стоящие после - превосходят либо равны).

// 'True' - figure is in right bottom quarter

{

int index = sort(SSPVect,

[](const Shape& a)

{

if (a.getPos().x>(X\_MAX/2) && a.getPos().y>(Y\_MAX/2)) return true;

return false;

});

cout << "Sorting container: predicate - figure's position in the left bottom quarter." << endl;

cout << "Sorted container: " << endl;

for (auto &i : SSPVect)

{

cout << \*i << " ";

}

cout << endl;

if (index>=0)

cout << "Index of first element matching predicate: " << index

<< endl << \*(SSPVect[index]);

else

cout << "No elements matching a predicate." << endl;

}

return 0;

}

**Figures.hpp**

#include "Shape.hpp"

#include "Pentagon.hpp"

#include "RegPentagon.hpp"

#include "Circle.hpp"

**vector.cpp**

#include <vector>

#include <iostream>

#include <assert.h>

#include <algorithm> // std::copy, std::rotate

#include <cstddef> // size\_t

#include <initializer\_list>

#include <stdexcept>

namespace stepik

{

template <typename Type>

class vector

{

public:

typedef Type\* iterator;

typedef const Type\* const\_iterator;

typedef Type value\_type;

typedef value\_type& reference;

typedef const value\_type& const\_reference;

typedef std::ptrdiff\_t difference\_type;

explicit vector(size\_t count = 0)

{

// std::cout << "Called count constructor" << std::endl;

// implement this

Type \*tmp = new Type[count];

m\_first = tmp;

m\_last = tmp + count;

}

template <typename InputIterator>

vector(InputIterator first, InputIterator last)

{

// std::cout << "Called iterators constructor" << std::endl;

// implement this

Type \*tmp = new Type[last-first];

std::copy(first, last, tmp);

m\_first = tmp;

m\_last = m\_first + (last-first);

}

vector(std::initializer\_list<Type> init)

{

// std::cout << "Called init\_list constructor" << std::endl;

// implement this

Type \*tmp = new Type[init.size()];

std::copy(init.begin(), init.end(), tmp);

m\_first = tmp;

m\_last = tmp + init.size();

}

vector(const vector& other) : vector(other.begin(), other.end())

{

// std::cout << "Called copy constructor" << std::endl;

// implement this

}

vector(vector&& other)

{

// implement this

// std::cout << "Called move constructor" << std::endl;

m\_first = other.m\_first;

m\_last = other.m\_last;

other.m\_first = nullptr;

other.m\_last = nullptr;

}

~vector()

{

// std::cout << "Called destructor" << std::endl;

// implement this

delete [] m\_first;

}

//assignment operators

vector& operator=(const vector& other)

{

// implement this

if (&other == this) return \*this;

// std::cout << "Called operator=" << std::endl;

vector tmp{other};

delete [] m\_first;

m\_first = tmp.m\_first;

m\_last = tmp.m\_last;

tmp.m\_first = nullptr;

tmp.m\_last = nullptr;

return \*this;

}

vector& operator=(vector&& other)

{

// implement this

if (&other == this) return \*this;

// std::cout << "Called move operator=" << std::endl;

delete [] m\_first;

m\_first = other.m\_first;

m\_last = other.m\_last;

other.m\_first = nullptr;

other.m\_last = nullptr;

return \*this;

}

// assign method

template <typename InputIterator>

void assign(InputIterator first, InputIterator last)

{

// implement this

vector tmp{first, last};

delete [] m\_first;

m\_first = tmp.m\_first;

m\_last = tmp.m\_last;

tmp.m\_first = nullptr;

tmp.m\_last = nullptr;

}

// resize methods

void resize(size\_t count)

{

// implement this

Type\* tmp = new Type[count];

delete [] m\_first;

m\_first = tmp;

m\_last = tmp + count;

}

//erase methods

iterator erase(const\_iterator pos)

{

// implement this

return erase(pos, pos+1);

}

iterator erase(const\_iterator first, const\_iterator last)

{

// implement this

if (first == m\_last) return m\_last;

// assert(first < last);

// assert((first>=m\_first) && (first < m\_last));

// assert((last<=m\_last) && (last>m\_first));

size\_t diff = last - first ;

size\_t tmp\_size = size() - diff;

size\_t ret\_pos = first - m\_first;

Type\* tmp = new Type[tmp\_size];

for (size\_t i = 0; i != ret\_pos; ++i)

{

tmp[i] = this->at(i);

}

for (size\_t i = ret\_pos+diff; i != size(); ++i)

{

tmp[i-diff] = this->at(i);

}

delete [] m\_first;

m\_first = tmp;

m\_last = tmp + tmp\_size;

tmp = nullptr;

return m\_first + ret\_pos;

}

//insert methods

iterator insert(const\_iterator pos, const Type& value)

{

return insert(pos, &value, 1+&value);

}

template <typename InputIterator>

iterator insert(const\_iterator pos, InputIterator first, InputIterator last)

{

// implement this

size\_t in\_size = last - first;

size\_t int\_pos = pos - m\_first;

size\_t new\_size = size() + in\_size;

Type\* tmp = new Type[new\_size];

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

{

tmp[i] = this->at(i);

}

for (size\_t i = int\_pos; i < int\_pos + in\_size; ++i)

{

tmp[i] = \*(first+i-int\_pos);

}

for (size\_t i = int\_pos + in\_size; i < new\_size; ++i)

{

tmp[i] = this->at(i-in\_size);

}

delete [] m\_first;

m\_first = tmp;

m\_last = tmp + new\_size;

return begin() + int\_pos;

}

//push\_back methods

void push\_back(const value\_type& value)

{

// implement this

size\_t new\_size = size() + 1;

Type\* tmp = new Type[new\_size];

std::copy(m\_first, m\_last, tmp);

tmp[size()] = value;

delete [] m\_first;

m\_first = tmp;

m\_last = tmp + new\_size;

}

//at methods

reference at(size\_t pos)

{

return checkIndexAndGet(pos);

}

const\_reference at(size\_t pos) const

{

return checkIndexAndGet(pos);

}

//[] operators

reference operator[](size\_t pos)

{

return m\_first[pos];

}

const\_reference operator[](size\_t pos) const

{

return m\_first[pos];

}

//\*begin methods

iterator begin()

{

return m\_first;

}

const\_iterator begin() const

{

return m\_first;

}

//\*end methods

iterator end()

{

return m\_last;

}

const\_iterator end() const

{

return m\_last;

}

//size method

size\_t size() const

{

return m\_last - m\_first;

}

//empty method

bool empty() const

{

return m\_first == m\_last;

}

void print()

{

for (auto i : \*this)

std::cout << i << " ";

std::cout << std::endl;

}

private:

reference checkIndexAndGet(size\_t pos) const

{

if (pos >= size())

{

throw std::out\_of\_range("out of range");

}

return m\_first[pos];

}

//your private functions

private:

iterator m\_first;

iterator m\_last;

};

}// namespace stepik

s**hared\_ptr.cpp**

#include <iostream>

// #define DEBUG

// #define DEBUG2

#include <memory>

#include <stdexcept>

namespace stepik

{

template <typename T>

class shared\_ptr

{

public:

template<class Y>

friend class shared\_ptr;

explicit shared\_ptr(T \*ptr = nullptr)

{

#ifdef DEBUG

std::cout << "Constructing shared " << this << " from ptr "

<< ptr << std::endl;

#endif

m\_ptr = ptr;

m\_count = nullptr;

if (m\_ptr)

{

m\_count = new long(1);

if (!m\_count) throw std::bad\_alloc{};

}

}

~shared\_ptr() noexcept

{

#ifdef DEBUG

std::cout << "Destroying shared\_ptr " << this << std::endl;

#endif

if (m\_ptr)

{

--(\*m\_count);

if ((\*m\_count) == 0)

{

#ifdef DEBUG

std::cout << "Counter is zero, deleting owned object "

<< m\_ptr << std::endl;

#endif

delete m\_ptr;

delete m\_count;

}

}

}

shared\_ptr(const shared\_ptr & other) noexcept

{

#ifdef DEBUG

std::cout << "Constructing shared\_ptr " << this << " from shared\_ptr "

<< &other << std::endl;

#endif

m\_ptr = other.m\_ptr;

m\_count = other.m\_count;

if (m\_ptr)

++(\*m\_count);

}

template<class Y>

shared\_ptr(const shared\_ptr<Y> & other) noexcept

{

#ifdef DEBUG

std::cout << "Constructing shared\_ptr " << this << " from shared\_ptr "

<< &other << std::endl;

#endif

m\_ptr = dynamic\_cast<T\*>(other.m\_ptr);

m\_count = other.m\_count;

if (m\_ptr)

++(\*m\_count);

}

shared\_ptr& operator=(const shared\_ptr &other) noexcept

{

#ifdef DEBUG

std::cout << "Assigning shared ptrs: " << this << " = "

<< &other << std::endl;

#endif

if ((void\*)this == (void\*)&other) return \*this;

if (m\_ptr)

{

--(\*m\_count);

if((\*m\_count)==0)

{

#ifdef DEBUG

std::cout << "Counter is zero, deleting owned object "

<< m\_ptr << std::endl;

#endif

delete m\_ptr;

delete m\_count;

m\_ptr = nullptr;

m\_count = nullptr;

}

}

m\_ptr = other.m\_ptr;

m\_count = other.m\_count;

if (m\_ptr)

++(\*m\_count);

return \*this;

}

template<class Y>

shared\_ptr& operator=(const shared\_ptr<Y> &other) noexcept

{

#ifdef DEBUG

std::cout << "Assigning shared ptrs: " << this << " = "

<< &other << std::endl;

#endif

if ((void\*)this == (void\*)&other) return \*this;

if (m\_ptr)

{

--(\*m\_count);

if((\*m\_count)==0)

{

#ifdef DEBUG

std::cout << "Counter is zero, deleting owned object "

<< m\_ptr << std::endl;

#endif

delete m\_ptr;

delete m\_count;

m\_ptr = nullptr;

m\_count = nullptr;

}

}

m\_ptr = dynamic\_cast<T\*>(other.m\_ptr);

m\_count = other.m\_count;

if (m\_ptr)

++(\*m\_count);

return \*this;

}

explicit operator bool() const noexcept

{

#ifdef DEBUG2

std::cout << "Bool called " << this << std::endl;

#endif

if (m\_ptr) return true;

return false;

}

T\* get() const noexcept

{

#ifdef DEBUG2

std::cout << "Get called " << this << std::endl;

#endif

return m\_ptr;

}

long use\_count() const noexcept

{

#ifdef DEBUG2

std::cout << "Count called " << this << std::endl;

#endif

if (!m\_count) return 0;

return \*m\_count;

}

T& operator\*() const noexcept

{

#ifdef DEBUG2

std::cout << "\* called " << this << std::endl;

#endif

return \*m\_ptr;

}

T\* operator->() const noexcept

{

#ifdef DEBUG2

std::cout << "-> called " << this << std::endl;

#endif

return m\_ptr;

}

void swap(shared\_ptr& x) noexcept

{

#ifdef DEBUG2

std::cout << "Swapping shared ptrs " << this << " and "

<< &x << std::endl;

#endif

T \*tmp\_ptr = m\_ptr;

long \*tmp\_count = m\_count;

m\_ptr = x.m\_ptr;

m\_count = x.m\_count;

x.m\_ptr = tmp\_ptr;

x.m\_count = tmp\_count;

}

void reset(T \*ptr = nullptr)

{

#ifdef DEBUG

std::cout << "Resetting shared\_ptr " << this << " from " << m\_ptr

<< " to " << ptr << std::endl;

#endif

if (m\_ptr)

{

--(\*m\_count);

if ((\*m\_count)==0)

{

#ifdef DEBUG

std::cout << "Counter is zero, deleting owned object "

<< m\_ptr << std::endl;

#endif

delete m\_ptr;

delete m\_count;

m\_ptr = nullptr;

m\_count = nullptr;

}

}

m\_ptr = ptr;

m\_count = nullptr;

if (m\_ptr)

{

m\_count = new long(1);

if (!m\_count) throw std::bad\_alloc{};

}

}

private:

// data members

T \*m\_ptr;

long \*m\_count;

};

// Comparison operators

template <class T, class U>

bool operator==(const shared\_ptr<T> &lhs,

const shared\_ptr<U> &rhs) noexcept

{

return (lhs.get() == rhs.get());

}

template <class T, class U>

bool operator!=(const shared\_ptr<T> &lhs,

const shared\_ptr<U> &rhs) noexcept

{

return !(lhs == rhs);

}

template <class T, class U>

bool operator<(const shared\_ptr<T> &lhs,

const shared\_ptr<U> &rhs) noexcept

{

return (lhs.get() < rhs.get());

}

template <class T, class U>

bool operator>(const shared\_ptr<T> &lhs,

const shared\_ptr<U> &rhs) noexcept

{

return (rhs < lhs);

}

template <class T, class U>

bool operator<=(const shared\_ptr<T> &lhs,

const shared\_ptr<U> &rhs) noexcept

{

return !(rhs < lhs);

}

template <class T, class U>

bool operator>=(const shared\_ptr<T> &lhs,

const shared\_ptr<U> &rhs) noexcept

{

return !(lhs < rhs);

}

// Class and nullptr

template <class T>

bool operator==(const shared\_ptr<T> &lhs, std::nullptr\_t rhs) noexcept

{

return !lhs;

}

template <class T>

bool operator==(std::nullptr\_t lhs, const shared\_ptr<T> &rhs) noexcept

{

return !rhs;

}

template <class T>

bool operator!=(const shared\_ptr<T> &lhs, std::nullptr\_t rhs) noexcept

{

return (bool)lhs;

}

template <class T>

bool operator!=(std::nullptr\_t lhs, const shared\_ptr<T> &rhs) noexcept

{

return (bool)rhs;

}

template <class T>

bool operator<(const shared\_ptr<T> &lhs, std::nullptr\_t rhs) noexcept

{

return lhs.get() < nullptr;

}

template <class T>

bool operator<(std::nullptr\_t lhs, const shared\_ptr<T> &rhs) noexcept

{

return nullptr < rhs.get();

}

template <class T>

bool operator>(const shared\_ptr<T> &lhs, std::nullptr\_t rhs) noexcept

{

return nullptr < lhs;

}

template <class T>

bool operator>(std::nullptr\_t lhs, const shared\_ptr<T> &rhs) noexcept

{

return rhs < nullptr;

}

template <class T>

bool operator<=(const shared\_ptr<T> &lhs, std::nullptr\_t rhs) noexcept

{

return !(nullptr < lhs);

}

template <class T>

bool operator<=(std::nullptr\_t lhs, const shared\_ptr<T> &rhs) noexcept

{

return !(rhs < nullptr);

}

template <class T>

bool operator>=(const shared\_ptr<T> &lhs, std::nullptr\_t rhs) noexcept

{

return !(lhs < nullptr);

}

template <class T>

bool operator>=(std::nullptr\_t lhs, const shared\_ptr<T> &rhs) noexcept

{

return !(nullptr < rhs);

}

// End of comparison operators

} // namespace stepik

**Shape.hpp**

#pragma once

#include "Vectors.hpp"

#include <iostream>

class Shape

{

public:

Shape() : Pos({0,0,0}), Angle(0), Color({0,0,0})

{

static int \_ID = 0;

++\_ID;

ID = \_ID;

}

virtual ~Shape() = default;

const V3i& getColor() const;

const V3d& getPos() const;

double getAngle() const;

int getID() const;

void setColor(const V3i& newColor);

virtual void setPos(const V3d& newPos);

void setAngle(double newAngle);

virtual void scale(double sc) = 0;

virtual void print() const = 0;

friend std::ostream& operator<<(std::ostream& ostr, const Shape& s);

protected:

V3d Pos;

double Angle;

V3i Color;

private:

int ID;

};

**Vectors.hpp**

#pragma once

class V3d

{

public:

double x, y, z;

V3d() : x(0), y(0), z(0)

{ }

V3d(double x, double y, double z = 0) : x(x), y(y), z(z)

{ }

};

class V5d

{

public:

double x, y, z, w, t;

V5d() : x(0), y(0), z(0), w(0), t(0)

{ }

V5d(double x, double y, double z, double w, double t) : x(x), y(y), z(z), w(w), t(t)

{ }

};

class V3i

{

public:

int x, y, z;

V3i() : x(0), y(0), z(0)

{ }

V3i(int x, int y, int z = 0) : x(x), y(y), z(z)

{ }

};

**Shape.cpp**

#include "Shape.hpp"

const V3i& Shape::getColor() const

{

return this->Color;

}

const V3d& Shape::getPos() const

{

return this->Pos;

}

double Shape::getAngle() const

{

return this->Angle;

}

int Shape::getID() const

{

return this->ID;

}

void Shape::setColor(const V3i& newColor)

{

this->Color = newColor;

this->Color.x %= 256;

this->Color.y %= 256;

this->Color.z %= 256;

}

void Shape::setPos(const V3d& newPos)

{

this->Pos = newPos;

}

void Shape::setAngle(double newAngle)

{

this->Angle = newAngle;

while (Angle < 0) Angle += 360;

while (Angle >= 360) Angle -= 360;

}

std::ostream& operator<<(std::ostream& ostr, const Shape& s)

{

s.print();

return ostr;

}

**Circle.hpp**

#pragma once

#include "Shape.hpp"

#include <iostream>

class Circle : public Shape

{

public:

Circle(double R = 0) : R(R)

{ }

double getRadius() const;

void setRadius(double newR);

void scale(double sc) override;

void print() const override;

private:

double R;

};

**Circle.cpp**

#include "Circle.hpp"

double Circle::getRadius() const

{

return this->R;

}

void Circle::setRadius(double newR)

{

this->R = newR;

}

void Circle::print() const

{

std::cout << "Circle object with ID " << getID() << std::endl;

std::cout << "It's position: " << Pos.x << " " << Pos.y << " " << Pos.z << std::endl;

std::cout << "It's color: " << Color.x << " " << Color.y << " " << Color.z << " " << std::endl;

std::cout << "It's angle: " << Angle << std::endl;

std::cout << "It's radius: " << R << std::endl;

}

void Circle::scale(double sc)

{

R\*=sc;

}

**Pentagon.hpp**

#pragma once

#include "Shape.hpp"

#include <iostream>

class Pentagon : public Shape

{

public:

const V5d& getDotsRadius() const;

const V5d& getDotsAngle() const;

void setDotsRadius(const V5d& newR);

void setDotsAngle(const V5d& newA);

void scale(double sc) override;

void print() const override;

private:

V5d R;

V5d A;

};

**Pentagon.cpp**

#include "Pentagon.hpp"

void Pentagon::print() const

{

std::cout << "Pentagon object with ID " << getID() << std::endl;

std::cout << "It's position: " << Pos.x << " " << Pos.y << " " << Pos.z << std::endl;

std::cout << "It's color: " << Color.x << " " << Color.y << " " << Color.z << " " << std::endl;

std::cout << "It's angle: " << Angle << std::endl;

std::cout << "It's points relative to the position of figure in polar(first is angle): " << std::endl;

std::cout << "(" << A.x << ", "<< R.x << ")" << ", "

<< "(" << A.y << ", "<< R.y << ")" << ", "

<< "(" << A.z << ", "<< R.z << ")" << ", "

<< "(" << A.w << ", "<< R.w << ")" << ", "

<< "(" << A.t << ", "<< R.t << ")" << std::endl;

}

void Pentagon::scale(double sc)

{

R.x\*=sc;

R.y\*=sc;

R.z\*=sc;

R.w\*=sc;

R.t\*=sc;

}

const V5d& Pentagon::getDotsRadius() const

{

return R;

}

const V5d& Pentagon::getDotsAngle() const

{

return A;

}

void Pentagon::setDotsRadius(const V5d& newR)

{

R = newR;

}

void Pentagon::setDotsAngle(const V5d& newA)

{

A = newA;

while (A.x < 0) A.x += 360;

while (A.y < 0) A.y += 360;

while (A.z < 0) A.z += 360;

while (A.w < 0) A.w += 360;

while (A.t < 0) A.t += 360;

while (A.x >= 360) A.x -= 360;

while (A.y >= 360) A.y -= 360;

while (A.z >= 360) A.z -= 360;

while (A.w >= 360) A.w -= 360;

while (A.t >= 360) A.t -= 360;

}

**RegPentagon.hpp**

#pragma once

#include "Shape.hpp"

class RegPentagon : public Shape

{

public:

double getDotsRadius() const;

void setDotsRadius(double newR);

void print() const override;

void scale(double sc) override;

private:

double R;

};

**RegPentagon.cpp**

#include "RegPentagon.hpp"

double RegPentagon::getDotsRadius() const

{

return R;

}

void RegPentagon::setDotsRadius(double newR)

{

R = newR;

}

void RegPentagon::print() const

{

std::cout << "Reg pentagon object with ID " << getID() << std::endl;

std::cout << "It's position: " << Pos.x << " " << Pos.y << " " << Pos.z << std::endl;

std::cout << "It's color: " << Color.x << " " << Color.y << " " << Color.z << " " << std::endl;

std::cout << "It's angle: " << Angle << std::endl;

std::cout << "It's radius: " << R << std::endl;

}

void RegPentagon::scale(double sc)

{

R \*= sc;

}