**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. 6304 |  | Виноградов К. А. |
| Преподаватель |  | Терентьев А. О. |

Санкт-Петербург

2018

**Цель работы**

Объединить предыдущие работы в приложении, использующем логику полиморфного хранения объектов.

**Постановка задачи**

Сгенерировать контейнер из 1000 фигур класса Shape, которые хранятся как shared\_ptr<Shape>, и применит к ним 2 стандартных алгоритма.

Алгоритмы:

1. Поиск смежных элементов, равных между собой по определенному критерию.
2. Составление упорядоченной группы из элементов двух выбранных диапазонов.

**Ход работы.**

Для реализации контейнера добавим наработки из предыдущих лабораторных работ. Создадим новый файл additional\_func.cpp и реализуем в нем требуемые алгоритмы. Тестирование подготовим в main.cpp.

Проведем тестирование. При тестировании функция assert() не выдала предупреждений, а значит результаты соответствуют ожиданиям.

**Выводы.**

В ходе данной лабораторной работы были объединены все предыдущие работы, а также закреплены принципы ООП.

Приложение А

shape.h

#pragma once

#define \_USE\_MATH\_DEFINES

#include <cmath>

#include <algorithm>

#include <iostream>

#include <cstdint>

#include <vector>

struct point

{

double x;

double y;

};

class Shape

{

public:

Shape(double ins\_x, double ins\_y)

{

scale = 1;

cen.x = ins\_x;

cen.y = ins\_y;

angle = 0;

colour = { 0, 0, 0 };

}

double getCenx()

{

return cen.x;

}

double getCeny()

{

return cen.y;

}

double toDeg(double rad)

{

return rad \* 180 / M\_PI;

}

double toRad(double deg)

{

return deg \* M\_PI / 180;

}

void setCenter(double ins\_x, double ins\_y)

{

for (auto& p : arr)

{

p.x += ins\_x - cen.x;

p.y += ins\_y - cen.y;

}

this->cen.x = ins\_x;

this->cen.y = ins\_y;

}

virtual void setAngle(double ins\_angle)

{

if (ins\_angle == 360 || ins\_angle == 0)

return;

angle += ins\_angle;

angle = std::fmod(angle, 360);

double temp\_x;

double rang = toRad(ins\_angle);

for (auto& p : arr)

{

temp\_x = p.x;

p.x = cen.x + (temp\_x - cen.x)\*cos(rang) - (p.y - cen.y)\*sin(rang);

p.y = cen.y + (temp\_x - cen.x)\*sin(rang) + (p.y - cen.y)\*cos(rang);

}

}

virtual void setScale(double ins\_scale)

{

scale \*= ins\_scale;

for (auto& p : arr)

{

p.x = (p.x - cen.x) \* ins\_scale + cen.x;

p.y = (p.y - cen.y) \* ins\_scale + cen.y;

}

}

void setColour(uint8\_t r, uint8\_t g, uint8\_t b)

{

colour = { r, g, b };

}

virtual void shape\_clear()

{

cen.x = cen.y = angle = 0;

scale = 1;

colour = { 0, 0, 0 };

arr.clear();

}

virtual void print(std::ostream& out)

{

out << "Class: shape" << '\n';

out << "Center: " << cen.x << ',' << cen.y << '\n';

out << "Angle: " << angle << '\n';

out << "Scale: " << scale << '\n';

out << "RGB: " << colour << '\n';

out << "Point number: " << arr.size() << '\n';

for (auto p : arr)

out << "x: " << p.x << " y: " << p.y << '\n';

out << '\n';

}

friend std::ostream& operator<<(std::ostream& out, Shape& in)

{

in.print(out);

return out;

}

protected:

struct RGB

{

uint8\_t R;

uint8\_t G;

uint8\_t B;

friend std::ostream& operator<<(std::ostream& out, RGB& in)

{

out << (unsigned int)in.R << ' ' << (unsigned int)in.G << ' ' << (unsigned int)in.B << '\n';

return out;

}

} colour;

point cen;

double angle;

double scale;

std::vector<point> arr;

};

class Ellipse : public Shape

{

public:

Ellipse(double ins\_x, double ins\_y, double ins\_foc, double rad\_sml, double rad\_lrg)

:Shape(ins\_x, ins\_y)

{

foc\_rang = 2 \* std::abs(ins\_foc);

l\_foc.x = ins\_x - std::abs(ins\_foc);

l\_foc.y = ins\_y;

r\_foc.x = ins\_x + std::abs(ins\_foc);

r\_foc.y = ins\_y;

point dot;

//Left edge

dot.x = cen.x - std::abs(rad\_lrg);

dot.y = cen.y;

arr.push\_back(dot);

//Right edge

dot.x = cen.x + std::abs(rad\_lrg);

dot.y = cen.y;

arr.push\_back(dot);

//Top edge

dot.x = cen.x;

dot.y = cen.y + std::abs(rad\_sml);

arr.push\_back(dot);

//Bottom edge

dot.x = cen.x;

dot.y = cen.y - std::abs(rad\_sml);

arr.push\_back(dot);

}

void print(std::ostream& out)

{

out << "Class: ellipse" << '\n';

out << "Center: " << cen.x << ',' << cen.y << '\n';

out << "Focus range: " << foc\_rang << "\n";

out << "Left focus: x: " << l\_foc.x << " y: " << l\_foc.y << '\n';

out << "Right focus: x: " << r\_foc.x << " y: " << r\_foc.y << '\n';

out << "Angle: " << angle << '\n';

out << "Scale: " << scale << '\n';

out << "RGB: " << colour << '\n';

out << "Point number: " << arr.size() << '\n';

for (auto p : arr)

out << "x: " << p.x << " y: " << p.y << '\n';

out << '\n';

}

void setAngle(double ins\_angle)

{

if (ins\_angle == 360 || ins\_angle == 0)

return;

angle += ins\_angle;

angle = std::fmod(angle, 360);

double temp\_x;

double rang = toRad(ins\_angle);

temp\_x = l\_foc.x;

l\_foc.x = cen.x + (temp\_x - cen.x)\*cos(rang) - (l\_foc.y - cen.y)\*sin(rang);

l\_foc.y = cen.y + (temp\_x - cen.x)\*cos(rang) + (l\_foc.y - cen.y)\*sin(rang);

temp\_x = r\_foc.x;

r\_foc.x = cen.x + (temp\_x - cen.x)\*cos(rang) - (r\_foc.y - cen.y)\*sin(rang);

r\_foc.y = cen.y + (temp\_x - cen.x)\*cos(rang) + (r\_foc.y - cen.y)\*sin(rang);

for (auto& p : arr)

{

temp\_x = p.x;

p.x = cen.x + (temp\_x - cen.x)\*cos(rang) - (p.y - cen.y)\*sin(rang);

p.y = cen.y + (temp\_x - cen.x)\*sin(rang) + (p.y - cen.y)\*cos(rang);

}

}

void setScale(double ins\_scale)

{

scale \*= ins\_scale;

foc\_rang \*= ins\_scale;

for (auto& p : arr)

{

p.x = (p.x - cen.x) \* ins\_scale + cen.x;

p.y = (p.y - cen.y) \* ins\_scale + cen.y;

}

}

void shape\_clear()

{

cen.x = cen.y = l\_foc.x = l\_foc.y = r\_foc.x = r\_foc.y = angle = foc\_rang = 0;

scale = 1;

colour = { 0, 0, 0 };

arr.clear();

}

protected:

point l\_foc;

point r\_foc;

double foc\_rang;

};

class Trap : public Shape

{

public:

Trap(double ins\_x, double ins\_y, double height\_length, double top\_base\_length, double bottom\_base\_length, double top\_base\_offset)

: Shape(ins\_x, ins\_y), height(std::abs(height\_length)), top\_base(std::abs(top\_base\_length)), bot\_base(std::abs(bottom\_base\_length))

{

point dot;

//Top left

dot.x = cen.x - top\_base / 2 + top\_base\_offset;

dot.y = cen.y + height / 2;

arr.push\_back(dot);

//Top right

dot.x = cen.x + top\_base / 2 + top\_base\_offset;

dot.y = cen.y + height / 2;

arr.push\_back(dot);

//Bot left

dot.x = cen.x - bot\_base / 2;

dot.y = cen.y - height / 2;

arr.push\_back(dot);

//Bot right

dot.x = cen.x + bot\_base /2 ;

dot.y = cen.y - height / 2;

arr.push\_back(dot);

}

void setScale(double ins\_scale)

{

scale \*= ins\_scale;

top\_base \*= ins\_scale;

bot\_base \*= ins\_scale;

height \*= ins\_scale;

for (auto& p : arr)

{

p.x = (p.x - cen.x) \* ins\_scale + cen.x;

p.y = (p.y - cen.y) \* ins\_scale + cen.y;

}

}

void print(std::ostream& out)

{

out << "Class: trapezium" << '\n';

out << "Center: " << cen.x << ',' << cen.y << '\n';

out << "Top base length: " << top\_base << '\n';

out << "Bot base length: " << bot\_base << '\n';

out << "Height length: " << height << '\n';

out << "Angle: " << angle << '\n';

out << "Scale: " << scale << '\n';

out << "RGB: " << colour << '\n';

out << "Point number: " << arr.size() << '\n';

for (auto p : arr)

out << "x: " << p.x << " y: " << p.y << '\n';

out << '\n';

}

void shape\_clear()

{

cen.x = cen.y = angle = top\_base = bot\_base = height = 0;

scale = 1;

colour = { 0, 0, 0 };

arr.clear();

}

protected:

double top\_base;

double bot\_base;

double height;

};

Приложение Б

My\_vector.h

#pragma once

#include <assert.h>

#include <cstring>

#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)

{

m\_first = new Type[count];

m\_last = m\_first + count;

}

template <typename InputIterator>

vector(InputIterator first, InputIterator last)

{

ptrdiff\_t count = last - first;

m\_first = new Type[count];

for (ptrdiff\_t i = 0; i < count; i++, first++)

{

m\_first[i] = \*first;

}

m\_last = m\_first + count;

}

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

{

m\_first = new Type[init.size()];

size\_t i = 0;

for (auto elem : init)

{

m\_first[i++] = elem;

}

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

}

vector(const vector& other)

{

size\_t count = other.size();

m\_first = new Type[count];

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

{

m\_first[i] = other[i];

}

m\_last = m\_first + count;

}

void swap(vector& first, vector& second)

{

using std::swap;

swap(first.m\_first, second.m\_first);

swap(first.m\_last, second.m\_last);

}

vector(vector&& other)

{

swap(\*this, other);

}

~vector()

{

delete[] m\_first;

}

//assignment operators

vector& operator=(const vector& other)

{

size\_t count = other.size();

delete [] m\_first;

m\_first = new Type[count];

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

{

m\_first[i] = other.m\_first[i];

}

m\_last = m\_first + count;

return \*this;

}

vector& operator=(vector&& other)

{

swap(\*this, other);

}

// assign method

template <typename InputIterator>

void assign(InputIterator first, InputIterator last)

{

size\_t count = last - first;

delete[] m\_first;

m\_first = new Type[count];

for (size\_t i = 0; i < count; i++, first++)

{

m\_first[i] = \*first;

}

m\_last = m\_first + count;

}

// resize methods

void resize(size\_t count)

{

vector temp = \*this;

delete[] m\_first;

m\_first = new Type[count];

m\_last = m\_first + count;

count = count > temp.size() ? temp.size() : count;

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

{

m\_first[i] = temp.m\_first[i];

}

}

//erase methods

iterator erase(const\_iterator pos)

{

iterator temp = &m\_first[pos - m\_first];

if (pos >= m\_last)

{

return temp;

}

size\_t count = m\_last - temp - 1;

memmove(temp, temp+1, count\* sizeof(Type));

m\_last--;

return temp;

}

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

{

iterator temp\_f = &m\_first[first - m\_first];

if (first == last || first >= m\_last || last > m\_last)

{

return temp\_f;

}

if (first - last == 1)

{

temp\_f = erase(first);

return temp\_f;

}

iterator temp\_l = &m\_first[last - m\_first];

size\_t count = m\_last - temp\_l;

memmove(temp\_f, temp\_l, count \* sizeof(Type));

m\_last -= last - first;

return temp\_f;

}

//insert methods

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

{

size\_t count = size() + 1;

Type\* temp = new Type[count];

iterator iter = &m\_first[pos - m\_first];

iterator value\_pos = std::copy(m\_first, iter, temp);

\*(value\_pos) = value;

std::copy(iter, m\_last, value\_pos + 1);

delete[] m\_first;

m\_first = new Type[count];

std::copy(temp, temp + count, m\_first);

delete[] temp;

m\_last = m\_first + count;

return &m\_first[pos - m\_first];

}

template <typename InputIterator>

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

{

size\_t count = (pos - begin());

if (pos > m\_last || first == last)

{

return begin() + count;

}

vector temp (size() + (last - first));

std::copy(const\_iterator(begin()), pos, temp.begin());

std::copy(first, last, temp.begin() + count);

if (pos + 1 < end())

std::copy(pos, const\_iterator(end()), temp.begin() + count + (last - first));

temp.swap(temp, \*this);

return begin() + count;

}

//push\_back methods

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

{

vector temp = \*this;

size\_t count = size() + 1;

delete[] m\_first;

m\_first = new Type[count];

m\_last = m\_first + count;

for (auto & el : temp)

std::copy(temp.begin(), temp.end(), m\_first);

m\_first[count - 1] = value;

}

//print method

void print()

{

for (size\_t i = 0; i < this->size(); i++)

{

std::cout << this->at(i) << std::endl;

}

std::cout << "vector size " << this->size() << std::endl;

}

//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;

}

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

public:

iterator m\_first;

iterator m\_last;

};

}// namespace stepik

Приложение в

Shared\_ptr.h

#pragma once

#include <iostream>

namespace stepik

{

template <typename T>

class shared\_ptr

{

public:

explicit shared\_ptr(T \*ptr = 0)

:ptr(ptr), count(ptr ? new size\_t(1) : new size\_t(0))

{

}

~shared\_ptr()

{

if (ptr != 0) {

--(\*count);

}

if (use\_count() == 0) {

delete ptr;

}

}

shared\_ptr(const shared\_ptr & other)

: ptr(other.ptr), count(other.count)

{

if (get() != 0)

{

++(\*count);

}

}

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

{

shared\_ptr(other).swap(\*this);

return \*this;

}

template <class Third>

shared\_ptr(const shared\_ptr<Third> & other)

: ptr(other.get()), count(other.get\_count())

{

if (get() != 0)

++(\*count);

}

template <class Third>

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

{

shared\_ptr(other).swap(\*this);

return \*this;

}

explicit operator bool() const

{

if (ptr)

return true;

else

return false;

}

T\* get() const

{

return ptr;

}

size\_t\* get\_count() const

{

return count;

}

size\_t use\_count() const

{

if (ptr)

return \*count;

else

return 0;

}

T& operator\*() const

{

return \*ptr;

}

T\* operator->() const

{

return ptr;

}

void swap(shared\_ptr& x) noexcept

{

std::swap(ptr, x.ptr);

std::swap(count, x.count);

}

void reset(T \*ptr = 0)

{

shared\_ptr<T>(ptr).swap(\*this);

}

private:

T \* ptr;

size\_t\* count;

};

template <class First, class Second>

bool operator== (const shared\_ptr<First>& left, const shared\_ptr<Second>& right)

{

return left.get() == right.get();

}

} // namespace stepik

Приложение Г

additional\_func.h

#pragma once

#ifndef additional\_func\_h

#define additional\_func\_h

#define \_USE\_MATH\_DEFINES

#include "my\_shared\_ptr.h"

#include "my\_vector.h"

#include "shape.h"

using namespace stepik;

struct neighbours

{

size\_t first;

size\_t second;

};

struct dual\_un

{

size\_t num;

double val;

};

bool ident\_vect(vector<size\_t> a, vector<neighbours> b);

bool ident\_vect(vector<size\_t> a, vector<dual\_un> b);

vector<shared\_ptr<Shape>> test\_create(size\_t amount);

vector<shared\_ptr<Shape>> create\_arr(size\_t amount);

vector<neighbours> neigh\_search(vector<shared\_ptr<Shape>> &arr);

vector<dual\_un> dual\_diap\_union(size\_t fst\_dp\_beg, size\_t fst\_dp\_end, size\_t sec\_dp\_beg, size\_t sec\_dp\_end, vector<shared\_ptr<Shape>> &arr);

#endif // !additional\_func\_hpp

Приложение Д

additional\_func.cpp

#include "stdafx.h"

#define \_USE\_MATH\_DEFINES

#include "additional\_func.h"

#include <time.h>

bool CompReal(double a, double b)

{

return a < b;

}

bool ident\_vect(vector<size\_t> a, vector<neighbours> b)

{

size\_t j = 0;

for (size\_t i = 0; i < a.size(); i+=2, j++)

{

if (a[i] != b[j].first)

return false;

if (a[i + 1] != b[j].second)

return false;

}

return true;

}

bool ident\_vect(vector<size\_t> a, vector<dual\_un> b)

{

for (size\_t i = 0; i < a.size(); i++)

{

if (a[i] != b[i].num)

return false;

}

return true;

}

vector<shared\_ptr<Shape>> test\_create(size\_t amount)

{

vector<shared\_ptr<Shape>> arr(amount);

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

{

if (i % 2 == 0)

arr[i] = shared\_ptr<Shape>(new Ellipse(i+2, 0, i + 5, i + 7, i + 15));

else

arr[i] = shared\_ptr<Shape>(new Trap(i + 1, 1 + i, i + 10, i + 10, i + 10, i + 10));

}

return arr;

}

vector<shared\_ptr<Shape>> create\_arr(size\_t amount)

{

vector<shared\_ptr<Shape>> arr(amount);

srand(time(nullptr));

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

{

if (rand() % 2 == 0)

arr[i] = shared\_ptr<Shape>(new Ellipse(rand() % 10, rand() % 10, rand() % 3 + 1, rand() % 10, 15));

else

arr[i] = shared\_ptr<Shape>(new Trap(rand() % 10, rand() % 10, rand() % 8, rand() % 3, rand() % 8, rand() % 5));

}

return arr;

}

vector<neighbours> neigh\_search(vector<shared\_ptr<Shape>> &arr)

{

vector<neighbours> result;

neighbours temp;

for (size\_t i = 0; i < arr.size() - 1; i++)

{

if ((\*arr[i]).getCenx() == (\*arr[i + 1]).getCenx())

{

temp.first = i;

temp.second = i + 1;

result.push\_back(temp);

}

}

return result;

}

vector<dual\_un> dual\_diap\_union(size\_t fst\_dp\_beg, size\_t fst\_dp\_end, size\_t sec\_dp\_beg, size\_t sec\_dp\_end, vector<shared\_ptr<Shape>> &arr)

{

vector <dual\_un> result;

dual\_un temp;

if (!(fst\_dp\_beg < 0 || fst\_dp\_beg > arr.size() || fst\_dp\_end < 0 || fst\_dp\_end > arr.size() || fst\_dp\_end - fst\_dp\_beg <= 0))

{

for (size\_t i = fst\_dp\_beg; i < fst\_dp\_end; i++)

{

temp.num = i;

temp.val = (\*arr[i]).getCenx() \* (\*arr[i]).getCeny();

result.push\_back(temp);

}

}

if (!(sec\_dp\_beg < 0 || sec\_dp\_beg > arr.size() || sec\_dp\_end < 0 || sec\_dp\_end > arr.size() || sec\_dp\_end - sec\_dp\_beg <= 0))

{

for (size\_t i = sec\_dp\_beg; i < sec\_dp\_end; i++)

{

temp.num = i;

temp.val = (\*arr[i]).getCenx() \* (\*arr[i]).getCeny();

result.push\_back(temp);

}

std::sort(result.begin(), result.end(), [](const dual\_un &a, const dual\_un &b)

{

return CompReal(a.val, b.val);

});

}

return result;

}

Приложение Е

main.cpp

#include "stdafx.h"

#include "additional\_func.h"

#include "my\_shared\_ptr.h"

#include "my\_vector.h"

#include "shape.h"

int main()

{

size\_t choice = 0;

std::cout << "Choose the mod :\n 1 - Test mod\n 2 - Work mod" << std::endl;

std::cin >> choice;

switch (choice)

{

case 1:

{

system("CLS");

vector<shared\_ptr<Shape>> test\_array = test\_create(10);

vector<neighbours> test\_pairs = neigh\_search(test\_array);

vector<size\_t> expected\_vec;

for (size\_t i = 0; i < 10; i++)

expected\_vec.push\_back(i);

assert(ident\_vect(expected\_vec, test\_pairs) == true);

expected\_vec.resize(0);

for (size\_t i = 0; i < 10; i++)

expected\_vec.push\_back(10);

assert(ident\_vect(expected\_vec, test\_pairs) == false);

expected\_vec.resize(0);

for (int i = 0; i< test\_pairs.size(); i++)

{

std::cout << "First: " << test\_pairs[i].first << " Second: " << test\_pairs[i].second << std::endl;

}

std::cout << "Amount of elments : " << test\_pairs.size() << std::endl;

system("pause");

system("CLS");

vector<dual\_un> test\_sort\_arr = dual\_diap\_union(0, 5, 5, 10, test\_array);

for (size\_t i = 0; i < 10; i+=2)

expected\_vec.push\_back(i);

for (size\_t i = 1; i < 10; i += 2)

expected\_vec.push\_back(i);

assert(ident\_vect(expected\_vec, test\_sort\_arr) == true);

expected\_vec.resize(0);

for (size\_t i = 0; i < 10; i++)

expected\_vec.push\_back(10);

assert(ident\_vect(expected\_vec, test\_sort\_arr) == false);

expected\_vec.resize(0);

for (int i = 0; i< test\_sort\_arr.size(); i++)

{

std::cout << "Number: " << test\_sort\_arr[i].num << " Value: " << test\_sort\_arr[i].val << std::endl;

}

std::cout << "Amount of elments : " << test\_sort\_arr.size() << std::endl;

}

break;

case 2:

{

system("CLS");

vector<shared\_ptr<Shape>> array = create\_arr(1000);

vector<neighbours> pairs = neigh\_search(array);

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

{

std::cout << "First: " << pairs[i].first << " Second: " << pairs[i].second << std::endl;

}

std::cout << "Amount of elments : " << pairs.size() << std::endl;

system("pause");

system("CLS");

vector<dual\_un> sort\_arr = dual\_diap\_union(rand() % 20, rand() % 20 + 40, rand() % 100 + 540, rand() % 50 + 700, array);

for (int i = 0; i< sort\_arr.size(); i++)

{

std::cout << "Number: " << sort\_arr[i].num << " Value: " << sort\_arr[i].val << std::endl;

}

std::cout << "Amount of elments : " << sort\_arr.size() << std::endl;;

}

break;

default:

std::cout << "Chose an option" << std::endl;

}

system("pause");

return 0;

}