# Міністерство освіти і науки України

Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"

Факультет інформатики та обчислювальної техніки Кафедра інформатики та програмної інженерії

Звіт

з лабораторної роботи №4 з дисципліни

«Основи програмування-2.

Модульне програмування»

«Перевантаження операторів»

Варіант 29

Виконав студент ІП-11 Тарасьонок Дмитро Євгенович

(шифр, прізвище, ім'я, по батькові)

Перевірив Вітковська Ірина Іванівна

( прізвище, ім'я, по батькові)

Київ 2022

Мета: вивчити механізми створення класів з використанням перевантажених операторів (операцій).

Умова задачі: Розробити клас "Вектор", членами якого є сферичні координати вектора у просторі. Реалізувати для нього декілька конструкторів, геттери, метод обчислення йцого декартових координат. Перевантажити оператори: для збільшення полярного кута на вказану величину (у градусах), для перевірки неколінеарності векторів. Створити три вектори (В1, В2, В3), використовуючи різні конструктори. Збільшити полярний кут вектора В1 на вказану величину. Знайти декартові координати зміненого вектора В1. Перевірити колінеарніть векторів В2 і В3.

Постановка задачі: для виконання поставленої задачі знадобиться спроектувати спочатку клас TPoint для зберігання декартових координат у просторі (x; y; z), а потім клас Vector, що матиме 3 поля: радіус, полярний кут та азимутальний кут. Пропонується створити три конструктори: за полярними координатами, за декартовими та конструктор за замовчуванням, що генеруватиме випадковий вектор. Вектори вважаються колінеарними, якщо кожну координату в декартовій системі координат можна розділити на відповідну координату другого вектора й в результаті усіх ділень ми отримаємо одне й те ж саме число (коефіцієнт), що й пропонується використовувати при виконанні завдання. Важливо запрограмувати правильні формули переведення полярних координат у декартові й навпаки.

Програма мовою C++:

Файл main.cpp

/\*

\* 1 курс, 2 семестр, Основи програмування, лабораторна робота №4.

\*

\* Варіант 29

\* Розробити клас "Вектор", членами якого є сферичні координати вектора у

\* просторі. Реалізувати для нього декілька конструкторів, геттери, метод

\* обчислення йцого декартових координат. Перевантажити оператори:

\* "+=" для збільшення полярного кута на вказану величину (у градусах),

\* "!=" для перевірки неколінеарності векторів. Створити три вектори

\* (В1, В2, В3), використовуючи різні конструктори. Збільшити полярний кут

\* вектора В1 на вказану величину. Знайти декартові координати зміненого

\* вектора В1. Перевірити колінеарніть векторів В2 і В3.

\*/

#include <locale>

#include <Windows.h>

#include "Vector.h"

using std::locale;

using std::cout;

using std::cin;

using std::endl;

int main()

{

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

locale::global(locale("rus\_rus.1251"));

srand(time(nullptr));

TPoint p1;

cout << "Введіть декартові координати першого вектора у форматі x; y; z: ";

cin >> p1;

Vector v1(p1);

double dbR, dbPolar, dbAzimuthal;

char ch1, ch2;

cout << "Введіть полярні координати другого вектора у форматі радіус; полярний\_кут; азимутальний кут: ";

cin >> dbR >> ch1 >> dbPolar >> ch2 >> dbAzimuthal;

Vector v2(dbR, dbPolar, dbAzimuthal);

Vector v3;

cout << "Перший вектор:" << endl << v1 << endl << endl << "Другой вектор:" << endl << v2 << endl << endl << "Третій вектор:" << endl << v3 << endl;

double ddbPolar;

cout << endl << "На скільки градусів треба збільшити полярний кут першого вектора?: ";

cin >> ddbPolar;

v1 += ddbPolar;

cout << "Перший вектор:" << endl << v1 << endl << endl;

bool bNonCollinearity = v2 != v3;

if (bNonCollinearity)

{

cout << "Другий та третій вектори неколінеарні" << endl;

}

else

{

cout << "Другий та третій вектори колінеарні" << endl;

}

system("pause");

return 0;

}

Файл TPoint.h

#pragma once

#include <iostream>

using std::ostream;

using std::istream;

using std::endl;

struct TPoint

{

TPoint(double x = 0, double y = 0, double z = 0) :pX{ x }, pY{ y }, pZ{ z } {}

double pX, pY, pZ;

};

ostream& operator << (ostream& os, const TPoint& a\_p);

istream& operator >> (istream& is, TPoint& a\_p);

Файл TPoint.cpp

#include "TPoint.h"

ostream& operator << (ostream& os, const TPoint& a\_p)

{

return os << "(x; y; z): (" << a\_p.pX << "; " << a\_p.pY << "; " << a\_p.pZ << ')';

}

istream& operator >> (istream& is, TPoint& a\_p)

{

char ch1, ch2;

return is >> a\_p.pX >> ch1 >> a\_p.pY >> ch2 >> a\_p.pZ;

}

Файл Vector.h

#pragma once

#define \_USE\_MATH\_DEFINES

#include <cmath>

#include <limits>

#include "TPoint.h"

class Vector

{

public:

Vector(double a\_dbR, double a\_dbPolar, double a\_dbAzimuthal);

Vector(const TPoint& point);

Vector();

double getRadius() const { return m\_dbR; }

double getPolar() const { return m\_dbPolar; }

double getAzimuthal() const { return m\_dbAzimuthal; }

TPoint toCartesian() const;

Vector& operator +=(double a\_dbAngle);

bool operator == (const Vector& a\_v) const;

bool operator != (const Vector& a\_v) const;

friend ostream& operator << (ostream& os, const Vector& a\_v);

private:

double m\_dbR;

double m\_dbPolar;

double m\_dbAzimuthal;

};

bool equalDouble(double db1, double db2);

Файл Vector.cpp

#include "Vector.h"

Vector& Vector::operator +=(double a\_dbAngle)

{

m\_dbPolar += a\_dbAngle;

return \*this;

}

bool Vector::operator == (const Vector& a\_v) const

{

TPoint thisCartesian = toCartesian();

TPoint secondCartesian = a\_v.toCartesian();

double dX = thisCartesian.pX / secondCartesian.pX;

double dY = thisCartesian.pY / secondCartesian.pY;

double dZ = thisCartesian.pZ / secondCartesian.pZ;

return equalDouble(dX, dY) && equalDouble(dY, dZ);

}

bool Vector::operator != (const Vector& a\_v) const

{

return !(\*this == a\_v);

}

Vector::Vector(double a\_dbR, double a\_dbPolar, double a\_dbAzimuthal)

{

if (a\_dbR < 0)

{

a\_dbR = 0;

}

while (a\_dbPolar < 0)

{

a\_dbPolar = 180 + a\_dbPolar;

}

while (180 < a\_dbPolar)

{

a\_dbPolar -= 180;

}

while (a\_dbAzimuthal < 0)

{

a\_dbAzimuthal = 360 + a\_dbAzimuthal;

}

while (360 <= a\_dbAzimuthal)

{

a\_dbAzimuthal -= 360;

}

m\_dbR = a\_dbR;

m\_dbPolar = a\_dbPolar;

m\_dbAzimuthal = a\_dbAzimuthal;

}

Vector::Vector(const TPoint& point)

:Vector(

sqrt(pow(point.pX, 2) + pow(point.pY, 2) + pow(point.pZ, 2)),

180.f / M\_PI \* acos(point.pZ / sqrt(pow(point.pX, 2) + pow(point.pY, 2) + pow(point.pZ, 2))),

180.f / M\_PI \* atan(point.pY / point.pX)

)

{

}

Vector::Vector()

:m\_dbR{ static\_cast<double>(rand() % 1001) / 10.f },

m\_dbPolar{ static\_cast<double>(rand() % 1801) / 10.f },

m\_dbAzimuthal{ static\_cast<double>(rand() % 3600) / 10.f }

{

}

TPoint Vector::toCartesian() const {

double x = m\_dbR \* sin(m\_dbPolar \* M\_PI / 180) \* cos(m\_dbAzimuthal \* M\_PI / 180);

double y = m\_dbR \* sin(m\_dbPolar \* M\_PI / 180) \* sin(m\_dbAzimuthal \* M\_PI / 180);

double z = m\_dbR \* cos(m\_dbPolar \* M\_PI / 180);

return TPoint(x, y, z);

}

bool equalDouble(double db1, double db2)

{

return abs(db1 - db2) <= std::numeric\_limits<double>::epsilon();

}

ostream& operator<<(ostream& os, const Vector& a\_v)

{

return os << "Декартові координати" << endl

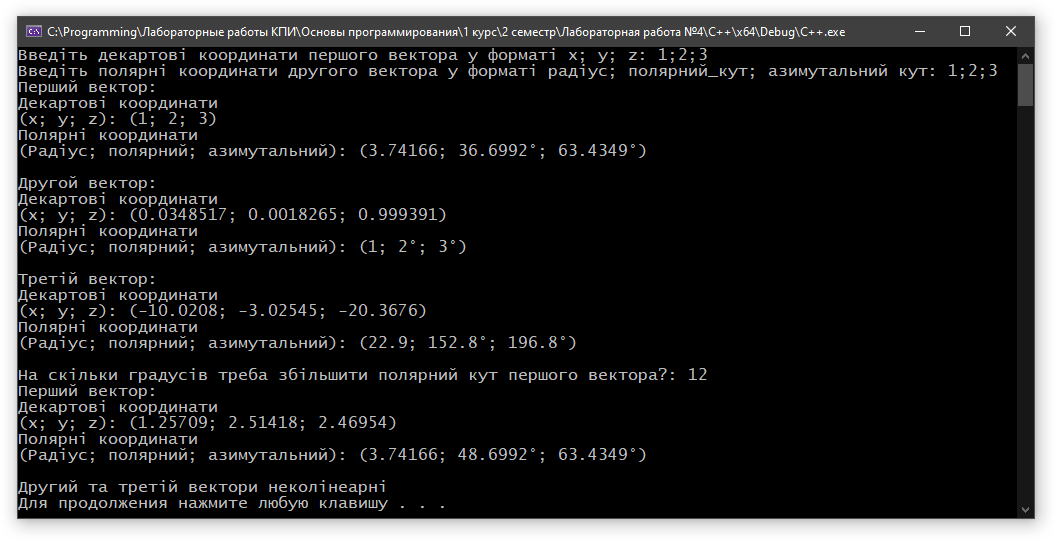
<< a\_v.toCartesian() << endl

<< "Полярні координати" << endl

<< "(Радіус; полярний; азимутальний): (" << a\_v.m\_dbR << "; " << a\_v.m\_dbPolar << "°; " << a\_v.m\_dbPolar << "°)";

}

Виконання коду мовою C++:



Висновок: під час виконання даної лабораторної роботи я отримав практичних навичок перевантаження операторів. На практиці перевантажив оператори +=, !=, ==, <<, >>, останні два з яких використовуються для введення/виведення й роблять код набагато красивішим. Також детальніше поглинувся в створення декількох конструкторів. Стикнувся з проблемою, що неможливо створити два конструктори з однаковою кількість та однаковими типами параметрів, тому для вирішення цієї проблеми я створив структуру TPoint і в конструктор Vector для полярних координат передавав безпосередні значення, а для декартових координат – об’єкт класу TPoint. Варто також зазначити, що під час виконання лабораторної роботи я стикнувся з проблемою з математичними функціями sin, cos, asin, acos та подібними, оскільки всі вони працюють із кутами в радіанах, а, згідно поставленої задачі, у спроектованому класі кути не треба було зберігати в радіанах, тому доводилося щоразу їх переводити.