Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»

Институт информационных технологий

Факультет компьютерных технологий

Кафедра информационных систем и технологий

Дисциплина: Конструирование программного обеспечения (КПО)

ОТЧЕТ

к лабораторной работе №1

на тему

**ПЕРЕГРУЗКА ФУНКЦИЙ И ОПЕРАТОРОВ**

Выполнил студент: гр. 181073

Наркевич Михаил Викторович

Руководитель:

Магистр технических наук Сицко В.А.

Минск 2022

**Лабораторная работа №1**

**Вариант 8**

**Тема работы:** Перегрузка функций и операторов.

**Цель работы:** Создание консольной программы реализующей перегрузку функции (в том числе с параметрами по умолчанию), перегрузку унарных и бинарных операторов, (в том числе и с помощью дружественных функций), реализацию функции преобразования типов.

**Общие требования к выполнению работы**

1.     Рассмотреть теоретические сведения по теме лабораторной работы.

2.     Создать проект консольной программы.

3.     Создать базовый класс согласно заданию. Все данные класса описать в закрытой секции, все методы класса реализовать как открытые.

4.     Создать конструктор инициализации с параметрами по умолчанию. Остальные конструкторы и деструктор создать по необходимости.

5.     Перегрузить операторы-члены класса:

-- присвоение =;

-- индексация элемента [];

-- инициализация ();

-- унарные арифметические (++);

-- бинарные арифметические (+ или - ) и сравнения (> или < или ==), причем только для однотипных операндов;

6.     Перегрузить дружественные функции-операторы:

-- унарные арифметические (--);

-- бинарные арифметические (+ или - ) и сравнения (> или < или ==), причем для разных типов операндов;

-- вывода в поток (<<).

7.     Создать функции преобразования:

-- во встроенный тип (int, double, …) (например, для подсчета характеристик массива (max, min, avg, count));

-- в пользовательский тип (например, класс описывающий размерность массива).

8.     В головной функции создать несколько экземпляров базового класса. Провести вызовы перегруженных функций и операторов. Вывести результаты на экран.

9.     Отладить и выполнить полученную программу.

10.            Создать отчет, включающий задание, листинг программы и результаты ее выполнения.

**Варианты заданий**

8.     «Массив комплексных чисел».

**Код программы**

#include <iostream>

#include <string>

#include <complex>

using namespace std;

class BasicSize {

private:

int width;

int height;

public:

BasicSize(int width, int height) {

this->width = width;

this->height = height;

}

void display() {

cout << "width: " << this->width << ", height: " << this->height << endl << endl;

}

};

class BasicArray {

private:

std::complex< double > basicArray[2];

public:

BasicArray(std::complex< double > element1 = 2.0+2.0i, std::complex< double > element2 = 2.0 + 2.0i) {

this->basicArray[0] = element1;

this->basicArray[1] = element2;

}

std::complex< double > getValue() {

return this->basicArray[0] + this->basicArray[1];

}

void setValue(std::complex< double > element1 = 0, std::complex< double > element2 = 0) {

this->basicArray[0] = element1;

this->basicArray[1] = element2;

}

BasicArray& operator = (const BasicArray& other) {

cout << "Operator = work" << endl;

basicArray[0] = other.basicArray[0];

basicArray[1] = other.basicArray[1];

return \*this;

}

std::complex< double > operator [] (int index) {

cout << "Operator [] work" << endl;

return this->basicArray[index];

}

std::complex< double > operator () (int index) {

cout << "Operator () work" << endl;

return this->basicArray[index];

}

BasicArray& operator ++ () {

cout << "Operator ++ pre work" << endl;

basicArray[0] += 1;

basicArray[1] += 1;

return \*this;

}

BasicArray operator ++ (int) {

cout << "Operator ++ post work" << endl;

BasicArray prev = \*this;

++\* this;

return prev;

}

BasicArray operator + (BasicArray other) {

cout << "Operator + work for same types" << endl;

return BasicArray(this->basicArray[0] + other.basicArray[0], this->basicArray[1] + other.basicArray[1]);

}

bool operator > (BasicArray other) {

cout << "Operator > work for same types" << endl;

return this->basicArray[0].real() + this->basicArray[1].real() > other.basicArray[0].real() + other.basicArray[1].real();

}

bool operator < (BasicArray other) {

cout << "Operator < work for same types" << endl;

return this->basicArray[0].real() + this->basicArray[1].real() < other.basicArray[0].real() + other.basicArray[1].real();

}

BasicArray& operator -- ();

BasicArray operator -- (int);

friend BasicArray operator - (const BasicArray& number1, std::complex< double > number2);

friend BasicArray operator - (std::complex< double > number1, const BasicArray& number2);

friend bool operator > (BasicArray number1, std::complex< double > number2);

friend bool operator > (std::complex< double > number1, BasicArray number2);

friend bool operator < (BasicArray number1, std::complex< double > number2);

friend bool operator < (std::complex< double > number1, BasicArray number2);

friend ostream& operator<<(ostream& os, const BasicArray& other);

operator std::complex< double >() const {

return basicArray[0] + basicArray[1];

}

operator string() const {

return to\_string(basicArray[0].real() + basicArray[1].real());

}

explicit operator BasicSize() const {

int width = basicArray[0].real();

int height = basicArray[1].real();

return BasicSize(width, height);

}

};

BasicArray& BasicArray::operator -- () {

cout << "Operator -- pre work" << endl;

basicArray[0] -= 1;

basicArray[1] -= 1;

return \*this;

}

BasicArray BasicArray::operator -- (int) {

cout << "Operator -- post work" << endl;

BasicArray prev = \*this;

--\* this;

return prev;

}

BasicArray operator - (const BasicArray& number1, std::complex< double > number2) {

cout << "Operator - work for different types" << endl;

return BasicArray(number1.basicArray[0] - number2, number1.basicArray[1] - number2);

}

BasicArray operator - (std::complex< double > number1, const BasicArray& number2) {

cout << "Operator - work for different types" << endl;

return BasicArray(number1 - number2.basicArray[0], number1 - number2.basicArray[1]);

}

bool operator > (BasicArray number1, std::complex< double > number2) {

cout << "Operator > work for different types" << endl;

return number1.basicArray[0].real() + number1.basicArray[1].real() > number2.real();

}

bool operator > (std::complex< double > number1, BasicArray number2) {

cout << "Operator > work for different types" << endl;

return number1.real() > number2.basicArray[0].real() + number2.basicArray[1].real();

}

bool operator < (BasicArray number1, std::complex< double > number2) {

cout << "Operator < work for different types" << endl;

return number1.basicArray[0].real() + number1.basicArray[1].real() < number2.real();

}

bool operator < (std::complex< double > number1, BasicArray number2) {

cout << "Operator < work for different types" << endl;

return number1.real() < number2.basicArray[0].real() + number2.basicArray[1].real();

}

ostream& operator << (ostream& os, const BasicArray& other) {

cout << "Operator << work " << endl;

return os << other.basicArray[0] << other.basicArray[1];

}

int main() {

// Default constructor

cout << "-- Default constructor --" << endl;

BasicArray number1;

cout << "number1: " << number1.getValue() << endl << endl;

// Operator '='

cout << "-- Operator '=' --" << endl;

BasicArray number2(7);

number2 = number1;

cout << "number2: " << number2.getValue() << endl << endl;

// Operator '[]'

cout << "-- Operator '[]' --" << endl;

cout << "number1[0]: " << number1[0] << endl << endl;

// Operator '()'

cout << "-- Operator '()' --" << endl;

cout << "number2(2): " << number2(2) << endl << endl;

// Operator '++'

cout << "-- Operator '++' post --" << endl;

number2++;

cout << "number2: " << number2.getValue() << endl << endl;

cout << "-- Operator '++' pre --" << endl;

++number2;

cout << "number2: " << number2.getValue() << endl << endl;

// Operator '+'

cout << "-- Operator '+' --" << endl;

BasicArray number3 = number1 + number2;

cout << "number3: " << number3.getValue() << endl << endl;

// Operator '>'

cout << "-- Operator '>' --" << endl;

cout << "number1 > number2: " << (number1 > number2) << endl << endl;

// Operator '<'

cout << "-- Operator '<' --" << endl;

cout << "number1 < number2: " << (number1 < number2) << endl << endl;

// Operator '--'

cout << "-- Operator '--' post --" << endl;

number2--;

cout << "number2: " << number2.getValue() << endl << endl;

cout << "-- Operator '--' pre --" << endl;

--number2;

cout << "number2: " << number2.getValue() << endl << endl;

// Operator '-' for different types

cout << "-- Operator '-' for different types --" << endl;

BasicArray number4 = number1 - 2;

cout << "number4: " << number4.getValue() << endl << endl;

// Operator '>' for different types

cout << "-- Operator '>' for different types --" << endl;

cout << "number1 > 2: " << (number1 > 2) << endl;

cout << "3 > number1: " << (3 > number1) << endl << endl;

// Operator '<' for different types

cout << "-- Operator '<' for different types --" << endl;

cout << "number1 < 2: " << (number1 < 2) << endl;

cout << "1 < number1: " << (1 < number1) << endl << endl;

// Operator '<<'

cout << "-- Operator '<<' --" << endl;

cout << "number2: " << number2 << endl << endl;

// (int)

cout << "-- (int) --" << endl;

cout << "(int)number1: " << (std::complex< double >)number1 << endl << endl;

// (string)

cout << "-- (string) --" << endl;

cout << "(string)number1: " << (string)number1 << endl << endl;

// (BasicSize)

cout << "-- (BasicSize) --" << endl;

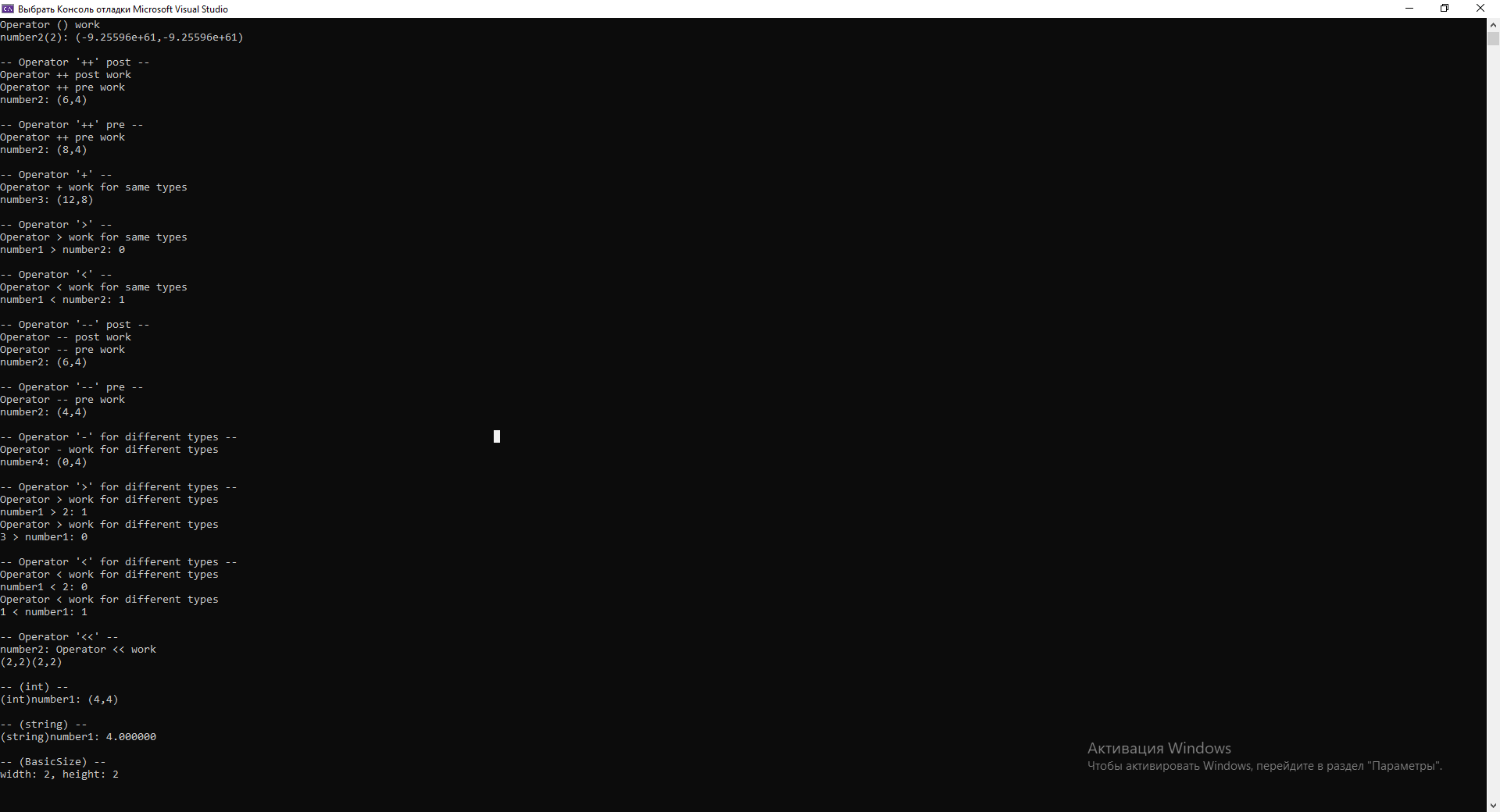
BasicSize basicSize = (BasicSize)number1;

basicSize.display();

return 0;

}

**Результат работы**



**Контрольные вопросы**

1.     Перегрузка функции.

Перегрузка функции — это определение нескольких функций с одинаковым именем, но различными параметрами. Параметры могут отличаться порядком следования, количеством, типом. Перегрузка функций нужна для того, чтобы избежать дублирования имён функций, выполняющих сходные действия, но с различной программной логикой.

Пример перегрузки:

void consoleLog(string value1) {  
 cout << value1;  
}  
  
void consoleLog(string value1, string valu2) {  
 cout << value1 << " " << value2;  
}  
  
int main() {  
 consoleLog("hello");  
 consoleLog("hello", "world");  
}

2.     Параметры функции по умолчанию.

Параметр по умолчанию (или «необязательный параметр») — это параметр функции, который имеет определенное значение (по умолчанию). Если пользователь не передает в функцию значение для параметра, то используется значение по умолчанию. Если же пользователь передает значение, то это значение используется вместо значения по умолчанию.

Пример параметров по умолчанию:

void consoleLog(string value1, string value2 = "default") {  
 cout << value1 << " " << value2;  
}  
  
int main() {  
 consoleLog("hello"); // hello default  
 consoleLog("hello", "world"); // hello world  
}

3.     Перегрузка унарных операторов.

Перегрузка унарных операторов — это возможность переопределить существующее поведение унарных операторов для работы с пользовательскими типами.

Пример перегрузки унарных операторов:

FractionalMatrix& operator ++ () {  
 cout << "Operator ++ pre work" << endl;  
  
 for (int i = 0; i < this->width; i++) {  
 for (int j = 0; j < this->height; j++) {  
 this->matrix[i][j] += 1;  
 }  
 }  
  
 return \*this;  
}

Теперь мы можем добавить к матрице 1, написав:

int main() {  
 FractionalMatrix fractionalMatrix(1, 1); // [1, 1]  
 fractionalMatrix++;  
 fractionalMatrix.showMatrix(); // [2, 2]  
}

4.     Перегрузка бинарных операторов.

Перегрузка бинарных операторов — это возможность переопределить существующее поведение бинарных операторов для работы с пользовательскими типами.

Пример перегрузки бинарных операторов:

FractionalMatrix operator + (FractionalMatrix other) {  
 cout << "Operator + work for same types" << endl;  
  
 float temporaryMatrix[1][2];  
  
 for (int i = 0; i < this->width; i++) {  
 for (int j = 0; j < this->height; j++) {  
 temporaryMatrix[i][j] = this->matrix[i][j] + other.matrix[i][j];  
 }  
 }  
  
 return FractionalMatrix(temporaryMatrix[0][0], temporaryMatrix[0][1]);  
}

Теперь мы можем сложить матрицы:

int main() {  
 FractionalMatrix fractionalMatrix1(1, 1); // [1, 1]  
 FractionalMatrix fractionalMatrix2(2, 2); // [2, 2]  
 FractionalMatrix fractionalMatrix3 = fractionalMatrix1 + fractionalMatrix2;  
 fractionalMatrix3.showMatrix(); // [3, 3]  
}

5.     Функции преобразования типов.

 Операторы преобразования (conversion operator) определяют преобразование объекта одного типа в другой.

Пример оператора преобразования:

operator float() const {  
 float sum = 0;  
  
 for (int i = 0; i < this->width; i++) {  
 for (int j = 0; j < this->height; j++) {  
 sum += this->matrix[i][j];  
 }  
 }  
  
 return sum;  
}

Пример оператора преобразования в пользовательский тип:

explicit operator BasicSize() const {  
 return BasicSize(this->width, this->height);  
}

Использование:

int main() {  
 FractionalMatrix fractionalMatrix(1, 3);  
  
 // (float)  
 cout << "(float)fractionalMatrix: " << (float)fractionalMatrix; // 4  
  
 // (BasicSize)  
 BasicSize basicSize = (BasicSize)fractionalMatrix;  
 basicSize.display(); // width: 1, height: 2  
}

**Вывод**

В ходе данной лабораторной работы я закрепил знания в написании: перегрузки операторов-членов класса (для однотипных операндов), перегрузки дружественных функций-операторов (для разных типов операндов), создал функции преобразования во встроенные и пользовательские типы.