**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ**

**КЫРГЫЗСКОЙ РЕСПУБЛИКИ**

**КЫРГЫЗСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ИМ. И. РАЗЗАКОВА**

**ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ**

**ОТЧЕТ**

**по лабораторной работе №5**

**Выполнила:** студент группы ПИ-3-21

Джанышбекова Акмарал

**Проверила:** Мусабаев Э. Б.

**Бишкек 2024**

**Задание №1**

Типы данных полезны там, где ошибки могут быть вызваны арифметическим переполнением, которое не допустимо.

Создайте и откомпилируйте класс **Int**. Перегрузите четыре бинарных целочисленных арифметических операции (+, -, \*, /) и унарные операции постфиксной и префиксной форм инкремента с помощью внутренней операторной функции так, чтобы их можно было использовать для операций с объектами класса **Int**.

Если результат какой-либо из них выходит за границы типа **int** (в 32-битной системе), имеющее значения от **2 147 483 648** до **-2 147 483 648**, то операция должна послать сообщение об ошибке и завершить программу. Для выявления ошибки арифметического переполнения используйте концепцию **исключения**.

Для облегчения проверки переполнения выполняйте вычисления с использованием типа **long** **double**. При описании унарных операций используйте указатель **this**.

**Код программы:**

#include <iostream>

#include <limits>

class Int {

private:

long double value;

public:

Int(long double initialValue) : value(initialValue) {}

// Бинарные операторы

// Оператор сложения

Int operator+(const Int& other) const {

CheckOverflow(value + other.value); // CheckOverflow - Проверка Переполнение

return Int(value + other.value);

}

// Оператор вычитания

Int operator-(const Int& other) const {

CheckOverflow(value - other.value);

return Int(value - other.value);

}

// Оператор умножение

Int operator\*(const Int& other) const {

CheckOverflow(value \* other.value);

return Int(value \* other.value);

}

// Оператор деление

Int operator/(const Int& other) const {

if (other.value == 0) {

std::cerr << "Ошибка: Деление на ноль недопустимо." << std::endl;

exit(EXIT\_FAILURE);

}

CheckOverflow(value / other.value);

return Int(value / other.value);

}

// Унарные операторы инкремента

Int& operator++() {

CheckOverflow(value + 1);

++value;

return \*this;

}

// Вывод текущего значения

void Display() const {

std::cout << "Текущее значение: " << static\_cast<int>(value) << std::endl;

}

private:

// Функция для проверки переполнения и завершения программы при необходимости

void CheckOverflow(long double result) const {

if (result > std::numeric\_limits<int>::max() || result < std::numeric\_limits<int>::min()) {

std::cerr << "Ошибка: Арифметическое переполнение. Результат не может быть представлен типом int." << std::endl;

exit(EXIT\_FAILURE);

}

}

};

int main() {

setlocale(LC\_ALL, "Ru");

Int num1 = 2147483647; // Максимальное значение int

Int num2 = 1;

std::cout << "Выполняем сложение:" << std::endl;

Int sum = num1 + num2;

sum.Display();

Int difference = num1 - num2;

difference.Display();

Int product = num1 \* num2;

product.Display();

Int quotient = num1 / num2;

quotient.Display();

Int result = ++num1;

result.Display();

return 0;

}

**Задание №2**

Опишите класс **fraction**, у которого есть одно закрытое целочисленное поле **chislo** типа **double**. Перегрузите для этого класса арифметические операции **сложения**, **вычитания**, **умножения** и **деления** так, чтобы они могли оперировать как с объектами класса, так и с числами (то есть выполнять, например, не только действие 3/4 +2/5, но и 1/2 + 4 или 2\* 5/6). Также перегрузите унарную операцию инкремента в префиксной или постфиксной форме увеличения дроби. Продемонстрируйте работу класса. Используйте конструктор по умолчанию и конструктор с одни аргументом для инициализации поля класса.

**Код программы:**

#include <iostream>

class Fraction {

private:

double chislo;

public:

// Конструктор по умолчанию

Fraction() : chislo(0.0) {}

// Конструктор с одним аргументом

Fraction(double value) : chislo(value) {}

// Перегрузка операторов сложения

Fraction operator+(const Fraction& other) const {

return Fraction(chislo + other.chislo);

}

Fraction operator+(double value) const {

return Fraction(chislo + value);

}

// Перегрузка операторов вычитания

Fraction operator-(const Fraction& other) const {

return Fraction(chislo - other.chislo);

}

Fraction operator-(double value) const {

return Fraction(chislo - value);

}

// Перегрузка операторов умножения

Fraction operator\*(const Fraction& other) const {

return Fraction(chislo \* other.chislo);

}

Fraction operator\*(double value) const {

return Fraction(chislo \* value);

}

// Перегрузка операторов деления

Fraction operator/(const Fraction& other) const {

if (other.chislo == 0.0) {

std::cerr << "Ошибка: Деление на ноль недопустимо." << std::endl;

exit(EXIT\_FAILURE);

}

return Fraction(chislo / other.chislo);

}

Fraction operator/(double value) const {

if (value == 0.0) {

std::cerr << "Ошибка: Деление на ноль недопустимо." << std::endl;

exit(EXIT\_FAILURE);

}

return Fraction(chislo / value);

}

// Перегрузка унарной операции инкремента (постфиксная форма)

Fraction operator++(int) {

Fraction temp = \*this;

chislo += 1.0;

return temp;

}

// Вывод текущего значения

void Display() const {

std::cout << "Текущее значение: " << chislo << std::endl;

}

};

int main() {

setlocale(LC\_ALL, "Ru");

Fraction frac1(3.0 / 4);

Fraction frac2(2.0 / 5);

std::cout << "Выполняем операцию сложения:" << std::endl;

std::cout << "Сложения: (3.0 / 4) + (2.0 / 5) = " << std::endl;

Fraction sum1 = frac1 + frac2;

sum1.Display();

std::cout << "\nВыполняем операцию сложения с числом:" << std::endl;

std::cout << "\nСложения с числом: (3.0 / 4) - 2.0 =" << std::endl;

Fraction sum2 = frac1 - 2.0;

sum2.Display();

std::cout << "\nВыполняем унарный инкремент:" << std::endl;

std::cout << "\nунарный инкремент: frac1++(3.0 / 4) =" << std::endl;

Fraction inc = frac1++;

inc.Display();

frac1.Display(); // После постфиксного инкремента значение frac1 также увеличивается

return 0;

}

**Задание №3**

Создать класс, в котором перегружается метод **rect\_area()**.

Этот метод возвращает площадь прямоугольника. В этой программе метод **rect\_area()** перегружается двумя способами. В первом — методу передаются оба размера фигуры. Эта версия используется для прямоугольника. Однако, в случае квадрата необходимо задавать только один аргумент, поэтому вызывается вторая версия метода **rect\_area().**

**Код программы:**

#include <iostream>

class Rectangle {

private:

double length;

double width;

public:

// Конструктор класса

Rectangle(double len, double wid) : length(len), width(wid) {}

// Перегруженный метод rect\_area() для прямоугольника

double rect\_area(double len, double wid) {

return len \* wid;

}

// Перегруженный метод rect\_area() для квадрата

double rect\_area(double side) {

return side \* side;

}

};

int main() {

setlocale(LC\_ALL, "Ru");

Rectangle rectangleObj(5.0, 3.0); // Прямоугольник

Rectangle squareObj(4.0, 4.0); // Квадрат

// Вызов первой версии метода rect\_area() для прямоугольника

std::cout << "Площадь прямоугольника: " << rectangleObj.rect\_area(5.0, 3.0) << std::endl;

// Вызов второй версии метода rect\_area() для квадрата

std::cout << "Площадь квадрата: " << squareObj.rect\_area(4.0) << std::endl;

return 0;

}

**ВОПРОСЫ К ЗАЩИТЕ ЛАБОРАТОРНОЙ РАБОТЫ №5**

1. Что такое полиморфизм?

**Полиморфизм** – это один интерфейс с множеством реализаций.

1. Что такое перегрузка операторов?

**Перегрузка операторов** – это возможность заставить компилятор осуществлять стандартные операции над нестандартными (пользовательскими) типами данных, тем самым оптимизировать код программы.

1. Что такое переопределение операторов?

**Переопределение операторов** - это предоставление новой реализации (определения) для существующих операторов в контексте классов или пользовательских типов данных.

1. Каков синтаксис операторной функции?

**Синтаксис операторной функции:**

class MyClass {

public:

Тип\_результата operator+(const MyClass& другой\_объект)

{

// Программный код

}

};

Пример для оператора сложения + в классе MyClass:

**Тип\_результата** - тип данных, который возвращает операторная функция.

**символ\_оператора** - символ оператора, который мы перегружаем.

**аргументы: тип\_и\_название** - аргументы, передаваемые в операторную функцию (тип и название каждого аргумента).

1. Какие операторы можно перегружать, а какие нельзя?

**Можно перегружать следующие операторы:**+ - \* / % ^ & | ~ ! = < >  
+= -= \*= /= %= ^= &= |=  
<< >> >>= <<= == != <= >=  
&& || ++ -- ->\* , -> [] ()  
new new[] delete delete[]

**Нельзя перегружать операторы:**

* + 1. :: (разрешение области видимости)

1. .  (доступ к членам класса)
2. .\* (выбор члена через указатель на член)
3. ?: тернарный оператор
4. Какова зависимость между количеством аргументов в операторной функции и количеством операндов? Объясните эту зависимость.

Количество аргументов в операторной функции соответствует количеству операндов, над которыми выполняется операция, и зависит от того, является ли оператор **бинарным** или **унарным**.

Зависимость между количеством аргументов в операторной функции и количеством операндов такова:

**для бинарных операторов** (например, сложение +, вычитание -), операторная функция принимает один аргумента, представляющих операнды, над которыми выполняется операция.

**Для унарных операторов** (например, инкремент ++, декремент --), операторная функция не имеет аргументов.

1. Каким образом перегруженные операции позволяют вид программного кода сделать более читабельным?

Перегруженные операции позволяют улучшить читаемость программного кода, делая его более лаконичным и понятным. Например, вместо длинного кода в точечной форме для сложения объектов, перегрузка операторов позволяет использовать более простой и естественный синтаксис, что делает код более легким для восприятия.

Cars.car3= Cars.car2+ Cars.car2;

car3= car1+ car2;