Министерство образования и науки Кыргызской Республики

Кыргызский государственный технический университет

им. И.Раззакова

Факультет информационных технологий

Кафедра «Программное обеспечение компьютерных систем»

Направление: 710400 «Программная инженерия»

Дисциплина: « Объектно-ориентрованное проектирование / Объектно-ориентрованное программирование»

**ОТЧЕТ**

По лабораторной работе №5

Тема: «Перегрузка операторов»

Выполнила: студентка группы

ПИ(б)-3-21 Пак Ксения

Проверил: Мусабаев Э. Б.

Бишкек – 2024

Задания

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**.

#include <iostream>

#include <limits>

class Int {

private:

int value;

public:

Int(int val) : value(val) {}

Int operator+(const Int& other) const {

CheckAddOverflow(value, other.value);

return Int(value + other.value);

}

Int operator-(const Int& other) const {

CheckSubtractOverflow(value, other.value);

return Int(value - other.value);

}

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

CheckMultiplyOverflow(value, other.value);

return Int(value \* other.value);

}

Int operator/(const Int& other) const {

if (other.value == 0) {

throw std::runtime\_error("Division by zero");

}

return Int(value / other.value);

}

Int& operator++() { // Префиксная форма

CheckIncrementOverflow(value);

++value;

return \*this;

}

Int operator++(int) { // Постфиксная форма

CheckIncrementOverflow(value);

Int temp(value);

++value;

return temp;

}

// Вспомогательные функции для проверки переполнения

static void CheckAddOverflow(int a, int b) {

if ((b > 0 && a > std::numeric\_limits<int>::max() - b) ||

(b < 0 && a < std::numeric\_limits<int>::min() - b)) {

throw std::overflow\_error("Arithmetic overflow");

}

}

static void CheckSubtractOverflow(int a, int b) {

if ((b < 0 && a > std::numeric\_limits<int>::max() + b) ||

(b > 0 && a < std::numeric\_limits<int>::min() + b)) {

throw std::overflow\_error("Arithmetic overflow");

}

}

static void CheckMultiplyOverflow(int a, int b) {

if (a > 0 && (b > std::numeric\_limits<int>::max() / a || b < std::numeric\_limits<int>::min() / a) ||

a < -1 && (b > std::numeric\_limits<int>::min() / a || b < std::numeric\_limits<int>::max() / a + 1)) {

throw std::overflow\_error("Arithmetic overflow");

}

}

static void CheckIncrementOverflow(int a) {

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

throw std::overflow\_error("Arithmetic overflow");

}

}

// Вывод значения

friend std::ostream& operator<<(std::ostream& os, const Int& num) {

os << num.value;

return os;

}

};

int main() {

try {

Int a(100);

Int b(200);

// Примеры использования перегруженных операторов

Int result1 = a + b;

Int result2 = a - b;

Int result3 = a \* b;

Int result4 = a / b;

std::cout << "a + b = " << result1 << std::endl;

std::cout << "a - b = " << result2 << std::endl;

std::cout << "a \* b = " << result3 << std::endl;

std::cout << "a / b = " << result4 << std::endl;

Int c(std::numeric\_limits<int>::max());

c++; // Попытка увеличить максимальное значение на 1

}

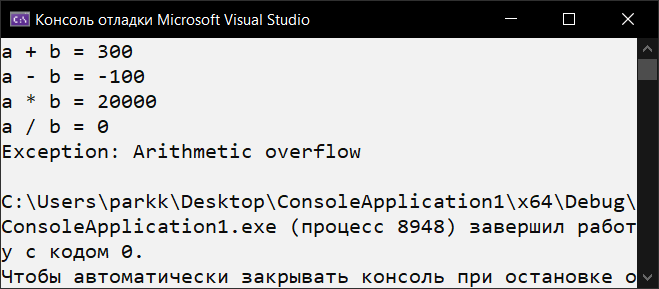
catch (const std::exception& e) {

std::cerr << "Exception: " << e.what() << std::endl;

}

return 0;

}



1. 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.

#include <iostream>

class Fraction {

private:

double chislo;

public:

Fraction() : chislo(0.0) {}

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

Fraction operator+(const Fraction& other) const {

return Fraction(chislo + other.chislo);

}

Fraction operator-(const Fraction& other) const {

return Fraction(chislo - other.chislo);

}

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

return Fraction(chislo \* other.chislo);

}

Fraction operator/(const Fraction& other) const {

if (other.chislo == 0.0) {

throw std::runtime\_error("Division by zero");

}

return Fraction(chislo / other.chislo);

}

// Перегрузка унарной операции инкремента (постфиксная форма)

Fraction operator++(int) {

Fraction temp(chislo);

chislo += 1.0;

return temp;

}

// Перегрузка унарной операции инкремента (префиксная форма)

Fraction& operator++() {

chislo += 1.0;

return \*this;

}

// Вывод значения

friend std::ostream& operator<<(std::ostream& os, const Fraction& fraction) {

os << fraction.chislo;

return os;

}

};

int main() {

// Пример использования класса Fraction

Fraction a(3.0);

Fraction b(2.5);

Fraction c = a + b;

Fraction d = a - b;

Fraction e = a \* b;

Fraction f = a / b;

std::cout << "a + b = " << c << std::endl;

std::cout << "a - b = " << d << std::endl;

std::cout << "a \* b = " << e << std::endl;

std::cout << "a / b = " << f << std::endl;

Fraction g = ++a; // Префиксная форма инкремента

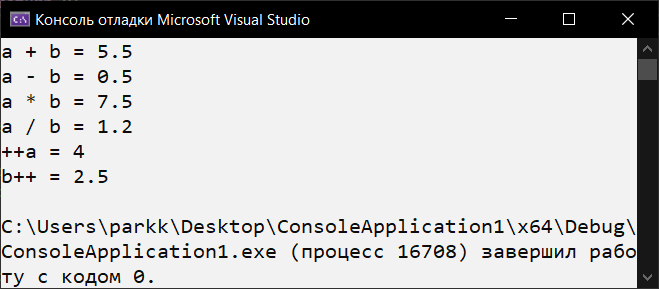
Fraction h = b++; // Постфиксная форма инкремента

std::cout << "++a = " << g << std::endl;

std::cout << "b++ = " << h << std::endl;

return 0;

}



1. 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().**

#include <iostream>

#include <Windows.h>

class Rectangle1 {

public:

double rect\_area(double a, double b) const {

return a \* b;

}

double rect\_area(double a) const {

return a \* a;

}

};

int main() {

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

double length, width, side;

std::cout << "Введите длину прямоугольника: ";

std::cin >> length;

std::cout << "Введите ширину прямоугольника: ";

std::cin >> width;

Rectangle1 rectangle;

std::cout << "Площадь прямоугольника: " << rectangle.rect\_area(length, width) << std::endl;

Rectangle1 square;

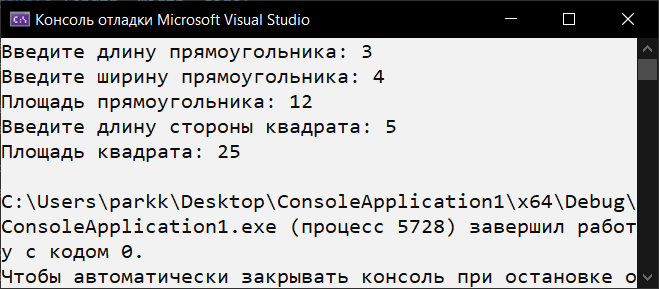
std::cout << "Введите длину стороны квадрата: ";

std::cin >> side;

std::cout << "Площадь квадрата: " << square.rect\_area(side) << std::endl;

return 0;

}



Ответы на вопросы

1. Что такое полиморфизм?

В языках программирования **полиморфизмом** называется способность функции обрабатывать данные разных типов. **Цель полиморфизма** - использование одного имени метода для задания общих для класса действий и тем самым снижение сложности программ. Пример полиморфизма в ООП – это перегрузка операторов.

1. Что такое перегрузка операторов?

**Перегрузка операторов** – это возможность заставить компилятор осуществлять стандартные операции над нестандартными (пользовательскими) типами данных.

1. Что такое переопределение операторов?

Переопределение операторов (operator overloading) - это механизм в некоторых языках программирования, который позволяет классам определять, какие операторы будут выполняться для их экземпляров. Вместо того чтобы ограничиваться стандартными операторами (например, +, -, \*, /), вы можете определить свою собственную логику для операций, когда объекты класса участвуют в них.

1. Каков синтаксис операторной функции?

Тип\_результата **operator** символ\_ оператора (аргументы: тип и название)

{

программный код

}

1. Какие операторы можно перегружать, а какие нельзя?

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

**Нельзя перегружать операторы:**

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

Перегруженной операции всегда требуется количество аргументов на один меньше, чем количество операндов, так как один из операндов является объектом, вызывающим операторную функцию. Поэтому для унарных операторов аргументы не нужны. А для бинарных нужен один аргумент.

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

Перегруженные операции позволяют использовать знаки и синтаксис языка программирования для пользовательских типов данных, делая код более читаемым и интуитивно понятным. Например, перегрузка операторов позволяет использовать стандартные арифметические операции для пользовательских классов, что делает код более выразительным и легким для восприятия.