**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ**

**КЫРГЫЗСКОЙ РЕСПУБЛИКИ**

**КЫРГЫЗСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ им. И.Раззакова**

**ФАКУЛЬТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ**

Кафедра: **Программное обеспечение компьютерных систем**

Курс «Объектно-ориентированное программирование»

**ОТЧЕТ**

**Лабораторная работа №5**

Выполнил: студент группы ПИ-3-21

Копжашаров Азамат

Проверил: Мусабаев Э.Б.

**Бишкек 2024**

# **Задание №1**

**Постановка задачи:**

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**.

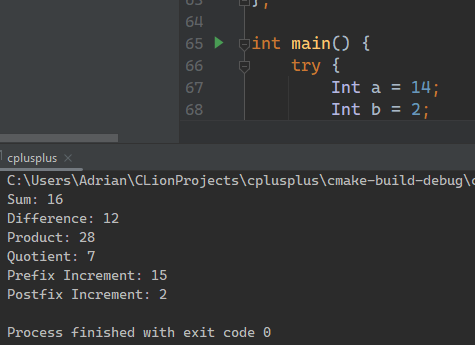
Для облегчения проверки переполнения выполняйте вычисления с использованием типа **long** **double**. При описании унарных операций используйте указатель **this**.

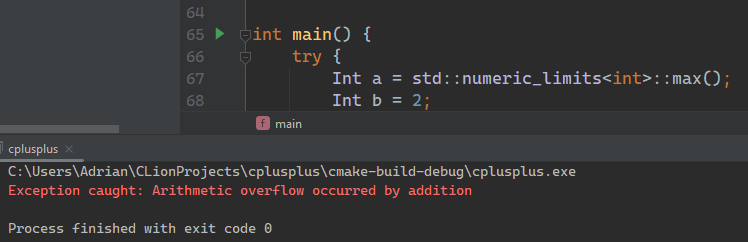
**Исходные данные:**

**Исходный код программы на C++:**

// 5.1  
#include <iostream>  
#include <limits>  
#include <exception>  
using namespace std;  
  
class Int{  
private:  
 int value;  
public:  
 Int(int val) : value(val) {}  
  
 Int operator+(const Int& other) const {  
 long double result = static\_cast<long double>(value) + other.value;  
 if(result < numeric\_limits<int>::min() || result > numeric\_limits<int>::max()){  
 throw overflow\_error("Arithmetic overflow occurred by addition");  
 }  
 return Int(static\_cast<int>(result));  
 }  
 Int operator-(const Int& other) const {  
 long double result = static\_cast<long double>(value) - other.value;  
 if(result < numeric\_limits<int>::min() || result > numeric\_limits<int>::max()){  
 throw overflow\_error("Arithmetic overflow occurred by subtraction");  
 }  
 return Int(static\_cast<int>(result));  
 }  
 Int operator\*(const Int& other) const {  
 long double result = static\_cast<long double>(value) \* other.value;  
 if(result < numeric\_limits<int>::min() || result > numeric\_limits<int>::max()){  
 throw overflow\_error("Arithmetic overflow occurred by multiplication");  
 }  
 return Int(static\_cast<int>(result));  
 }  
 Int operator/(const Int& other) const {  
 if(other.value == 0){  
 throw invalid\_argument("Division by zero");  
 }  
 return Int(value / other.value);  
 }  
  
 // prefix  
 Int& operator++(){  
 if(value == numeric\_limits<int>::max()){  
 throw overflow\_error("Arithmetic overflow occurred by prefix form");  
 }  
 ++value;  
 return \*this;  
 }  
 // postfix  
 Int operator++(int){  
 Int temp = \*this;  
 ++(\*this);  
 return temp;  
 }  
 // Output operator  
 friend ostream& operator<<(ostream& os, const Int& num) {  
 os << num.value;  
 return os;  
 }  
};  
  
int main() {  
 try {  
 Int a = std::numeric\_limits<int>::max();  
 Int b = 2;  
  
 // Test arithmetic operations  
 Int sum = a + b;  
 Int difference = a - b;  
 Int product = a \* b;  
 Int quotient = a / b;  
  
 // Test increment operators  
 ++a;  
 Int c = b++;  
  
 std::cout << "Sum: " << sum << std::endl;  
 std::cout << "Difference: " << difference << std::endl;  
 std::cout << "Product: " << product << std::endl;  
 std::cout << "Quotient: " << quotient << std::endl;  
 std::cout << "Prefix Increment: " << a << std::endl;  
 std::cout << "Postfix Increment: " << c << std::endl;  
 } catch (const exception& e) {  
 std::cerr << "Exception caught: " << e.what() << std::endl;  
 }  
 return 0;  
}  
/\*  
Int a = std::numeric\_limits<int>::max();  
 \*/

**Тесты:**

****

****

# **Задание №2**

**Постановка задачи:**

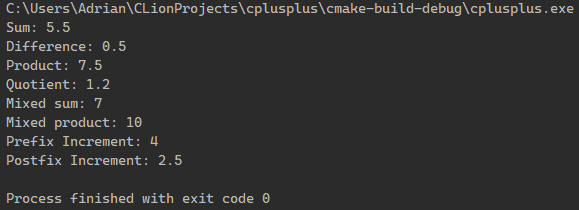
Опишите класс **fraction**, у которого есть одно закрытое целочисленное поле **chislo** типа **double**. Перегрузите для этого класса арифметические операции **сложения**, **вычитания**, **умножения** и **деления** так, чтобы они могли оперировать как с объектами класса, так и с числами (то есть выполнять, 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.

**Исходные данные:**

**Исходный код программы на C++:**

// 5.2  
#include <iostream>  
using namespace std;  
  
class Fraction {  
private:  
 double chislo;  
  
public:  
 // Constructors  
 Fraction() : chislo(0) {}  
 Fraction(double value) : chislo(value) {}  
  
 // Binary arithmetic operators  
 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) {  
 std::cerr << "Division by zero\n";  
 exit(1);  
 }  
 return Fraction(chislo / other.chislo);  
 }  
  
 // Unary increment operators  
 Fraction& operator++() { // Prefix increment  
 chislo++;  
 return \*this;  
 }  
  
 Fraction operator++(int) { // Postfix increment  
 Fraction temp = \*this;  
 ++(\*this);  
 return temp;  
 }  
  
 // Friend function to overload operator+ for mixed-mode arithmetic  
 friend Fraction operator+(const Fraction& fraction, double value) {  
 return Fraction(fraction.chislo + value);  
 }  
  
 friend Fraction operator\*(double value, const Fraction& fraction) {  
 return Fraction(value \* fraction.chislo);  
 }  
  
 // Output operator  
 friend ostream& operator<<(ostream& os, const Fraction& fraction) {  
 os << fraction.chislo;  
 return os;  
 }  
};  
  
int main() {  
 Fraction a(3.0), b(2.5);  
 double c = 4.0;  
  
 // Test arithmetic operations with fractions  
 Fraction sum = a + b;  
 Fraction difference = a - b;  
 Fraction product = a \* b;  
 Fraction quotient = a / b;  
  
 // Test mixed-mode arithmetic  
 Fraction mixed\_sum = a + c;  
 Fraction mixed\_product = c \* b;  
  
 // Test increment operators  
 ++a;  
 Fraction post\_increment = b++;  
  
 // Display results  
 cout << "Sum: " << sum << endl;  
 cout << "Difference: " << difference << endl;  
 cout << "Product: " << product << endl;  
 cout << "Quotient: " << quotient << endl;  
 cout << "Mixed sum: " << mixed\_sum << endl;  
 cout << "Mixed product: " << mixed\_product << endl;  
 cout << "Prefix Increment: " << a << endl;  
 cout << "Postfix Increment: " << post\_increment << endl;  
  
 return 0;  
}

**Тесты:**



# **Задание №3**

**Постановка задачи:**

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

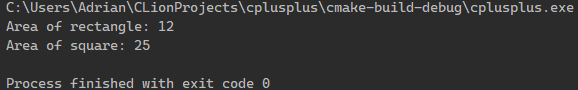
Этот метод возвращает площадь прямоугольника. В этой программе метод **rect\_area()** перегружается двумя способами. В первом — методу передаются оба размера фигуры. Эта версия используется для прямоугольника. Однако, в случае квадрата необходимо задавать только один аргумент, поэтому вызывается вторая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

**Исходные данные:**

**Исходный код программы на C++:**

// 5.3  
#include <iostream>  
using namespace std;  
  
class Rectangle {  
private:  
 double length;  
 double width;  
  
public:  
 Rectangle(double l, double w) : length(l), width(w) {}  
 Rectangle(double size) : length(size), width(size) {}  
  
 // Method to calculate area of rectangle  
 double rect\_area(double l, double w) {  
 return l \* w;  
 }  
  
 // Overloaded method for rectangle  
 double rect\_area() {  
 return length \* width;  
 }  
  
 // Overloaded method for square  
 double rect\_area(double side) {  
 return side \* side;  
 }  
};  
  
int main() {  
 Rectangle rectangle(4.0, 3.0);  
 Rectangle square(5.0);  
  
 // Calculate area of rectangle  
 cout << "Area of rectangle: " << rectangle.rect\_area() << endl;  
  
 // Calculate area of square  
 cout << "Area of square: " << square.rect\_area(5.0) << endl;  
 return 0;  
}

**Тесты:**



# **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.

**2. Что такое перегрузка операторов?**

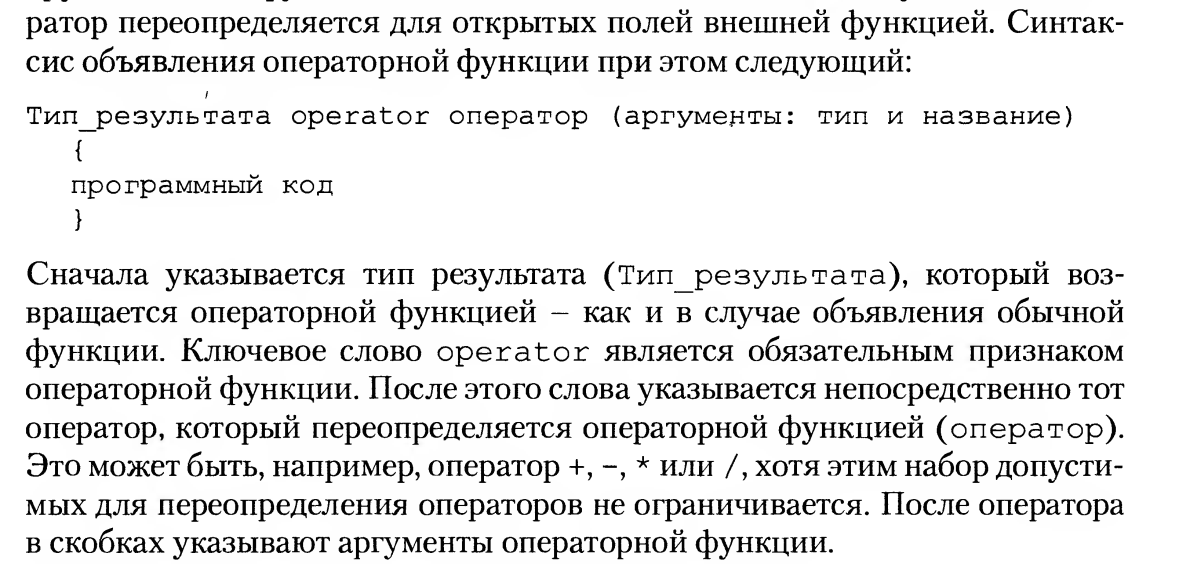
**Перегрузка операторов** – это возможность заставить компилятор осуществлять стандартные операции над нестандартными (пользовательскими) типами данных.

**3. Что такое переопределение операторов?**

**Перегрузка операторов в классах – это дополнительное переопределение стандартных операторов.**

Переопределить оператор для работы со стандартными типами на оператор для работы с пользовательскими типами можно с помощью операторной функции **operator**.

**4. Каков синтаксис операторной функции?**



**5. Какие операторы можно перегружать, а какие нельзя?**

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

**Нельзя перегружать операторы:**

:: (разрешение области видимости)

.  (доступ к членам класса)

.\* (выбор члена через указатель на член)

?: тернарный оператор

**6. Какова зависимость между количеством аргументов в операторной функции и количеством операндов? Объясните эту зависимость.**

**Правило 1**: при перегрузке бинарных операторов операторной функции передается только один правый от знака операции аргумент. Левый аргумент в операторе сложения объектов отождествляется с объектом, из которого вызывается операторная функция, поэтому он не включается в список аргументов ОФ.

**Правило 2**: при перегрузке унарных операторов операторной функции аргументы не передаются (за исключением перегрузки префиксной формы операторов инкремента и декремента, где в круглых скобках стоит признак для компилятора постфиксной формы оператора инкремента в виде (int)). Причина связана с тем, что один аргумент операторной функции – объект, из которого вызывается эта функция. Доступ к этому объекту можно получить через указатель this.

**Обобщим правила.** Перегруженной операции всегда требуется количество аргументов на один меньше, чем количество операндов. Это связано с тем, что один из объектов является объектом, вызывающим функцию.

По этой причине для унарных операций не нужны аргументы.

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

Мы уже говорили, что перегруженные операции позволяют *вид программного кода сделать более читабельным, компактным и понятным****.***

Например, код операции сложения объектов в виде в точечном формате:

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

Операторная функция дает возможность заменить этот код на более простой:

car3= car2+ car2;