**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ**

**КЫРГЫЗСКОЙ РЕСПУБЛИКИ**

**КЫРГЫЗСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ**

**УНИВЕРСИТЕТ им. И. РАЗЗАКОВА**

**Кафедра «Программное обеспечение компьютерных систем»**

**Отчет**

**ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ С++**

**Лабораторная работа №5**

**Выполнила: Эмилбекова Элиза ПИ-2-21**

**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**.

**КОД:**

#include <iostream>

#include <limits>

using namespace std;

class Int {

private:

long double value;

public:

// Конструктор класса

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

// Перегрузка оператора +

Int operator+(const Int& other) const {

return checkOverflow(value + other.value, "Addition");

}

// Перегрузка оператора -

Int operator-(const Int& other) const {

return checkOverflow(value - other.value, "Subtraction");

}

// Перегрузка оператора \*

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

return checkOverflow(value \* other.value, "Multiplication");

}

// Перегрузка оператора /

Int operator/(const Int& other) const {

if (other.value == 0) {

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

}

return checkOverflow(value / other.value, "Division");

}

// Перегрузка оператора префиксного инкремента

Int& operator++() {

value = checkOverflow(value + 1, "Increment").value;

return \*this;

}

// Перегрузка оператора постфиксного инкремента

Int operator++(int) {

Int temp = \*this;

++(\*this);

return temp;

}

// Метод для вывода значения

void display() const {

std::cout << value << endl;

}

private:

// Метод для проверки переполнения и выброса исключения при необходимости

Int checkOverflow(long double result, const std::string& operation) const {

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

throw std::overflow\_error("Arithmetic overflow in " + operation + " operation");

}

return Int(static\_cast<int>(result));

}

};

int main() {

setlocale(LC\_ALL, "Russian");

try {

int initialValue;

cout << "Введите начальное значение: ";

cin >> initialValue;

Int myInt(initialValue);

Int result = myInt + 5;

cout << "Результат сложения: ";

result.display();

result = myInt - 3;

cout << "Результат вычитания: ";

result.display();

result = myInt \* 2;

cout << "Результат умножения: ";

result.display();

result = myInt / 2;

cout << "Результат деления: ";

result.display();

cout << "Постфиксный инкремент: ";

(myInt++).display();

cout << "После постфиксного инкремента: ";

myInt.display();

cout << "Префиксный инкремент: ";

(++myInt).display();

}

catch (const std::exception& e) {

cerr << "Ошибка: " << e.what() << endl;

}

return 0;

}

Класс Int: Класс Int представляет пользовательский целочисленный тип данных. Его конструктор принимает начальное значение типа int и инициализирует поле value класса.

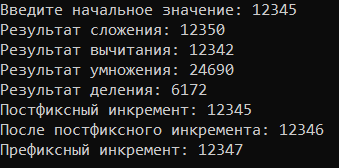
Бинарные операторы (+, -, \*, /): Класс перегружает эти операторы для выполнения арифметических операций с объектами класса Int. При выполнении каждой из операций происходит проверка на переполнение с использованием метода checkOverflow. Если результат операции выходит за границы типа int, выбрасывается исключение overflow\_error.

Унарные операторы инкремента (++): Класс также перегружает префиксный и постфиксный операторы инкремента для увеличения значения объекта на 1. При этом также используется проверка на переполнение.

Метод checkOverflow: Этот метод принимает результат арифметической операции и проверяет, не выходит ли он за границы типа int. Если да, выбрасывается исключение. В противном случае, результат приводится к типу int и возвращается в виде нового объекта Int.

Метод display: Простой метод для вывода значения объекта Int.

**Работа программы:**

****

**Все правильно вычисляется.**

****

Программа корректно обработала ситуацию арифметического переполнения при сложении с максимальным значением типа **int** (2147483647). Это означает, что программа успешно распознала потенциальное переполнение и выбросила исключение **overflow\_error**, предотвращая некорректные результаты.

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.**

**КОД:**

#include <iostream>

using namespace std;

class Fraction {

private:

double chislo;

public:

// Конструкторы

Fraction() : chislo(0.0) {}

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

// Методы для доступа к chislo

double getChislo() const {

return chislo;

}

void setChislo(double value) {

chislo = value;

}

// Перегрузка операторов сложения, вычитания, умножения и деления для объектов класса

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) {

return Fraction(chislo / other.chislo);

}

else {

std::cerr << "Error: Division by zero\n";

return Fraction();

}

}

// Перегрузка операторов сложения, вычитания, умножения и деления для работы с числами

Fraction operator+(double number) const {

return Fraction(chislo + number);

}

Fraction operator-(double number) const {

return Fraction(chislo - number);

}

Fraction operator\*(double number) const {

return Fraction(chislo \* number);

}

Fraction operator/(double number) const {

if (number != 0.0) {

return Fraction(chislo / number);

}

else {

cerr << "Error: Division by zero\n";

return Fraction();

}

}

// Перегрузка унарной операции инкремента (префиксная форма)

Fraction operator++() {

chislo += 1.0;

return \*this;

}

// Перегрузка унарной операции инкремента (постфиксная форма)

Fraction operator++(int) {

Fraction temp = \*this;

chislo += 1.0;

return temp;

}

// Метод для вывода значения дроби

void display() const {

cout << chislo << endl;

}

// Метод для ввода значения дроби с консоли

void input() {

cout << "Enter a fraction: ";

cin >> chislo;

}

};

// Перегрузка операторов сложения, вычитания, умножения и деления для работы с числами (внешние функции)

Fraction operator+(double number, const Fraction& fraction) {

return Fraction(number + fraction.getChislo());

}

Fraction operator-(double number, const Fraction& fraction) {

return Fraction(number - fraction.getChislo());

}

Fraction operator\*(double number, const Fraction& fraction) {

return Fraction(number \* fraction.getChislo());

}

Fraction operator/(double number, const Fraction& fraction) {

if (fraction.getChislo() != 0.0) {

return Fraction(number / fraction.getChislo());

}

else {

std::cerr << "Error: Division by zero\n";

return Fraction();

}

}

int main() {

Fraction a;

Fraction b;

a.input();

b.input();

Fraction result1 = a + b;

std::cout << "Sum: ";

result1.display();

Fraction result2 = a - b;

std::cout << "Difference: ";

result2.display();

Fraction result3 = a \* b;

std::cout << "Product: ";

result3.display();

Fraction result4 = a / b;

std::cout << "Quotient: ";

result4.display();

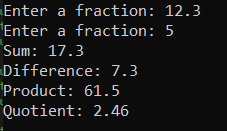
return 0;

}

Программа представляет собой простой класс Fraction, предназначенный для работы с дробными числами. Класс включает в себя операторы для сложения, вычитания, умножения и деления как между объектами класса, так и между объектами и числами. Для обеспечения инкапсуляции, поле chislo является закрытым и доступно только через методы getChislo и setChislo.

Программа также включает в себя перегруженные операторы для ввода и вывода дробных чисел с консоли.

**Работа программы:**

****

**Он работает так же и числами например тут 5 .**

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().**

**Код:**

include <iostream>

using namespace std;

class Rectangle {

public:

// Перегрузка метода rect\_area() для прямоугольника (длина и ширина)

double rect\_area(double length, double width) {

return length \* width;

}

// Перегрузка метода rect\_area() для квадрата (одна сторона)

double rect\_area(double side) {

return side \* side;

}

};

int main() {

Rectangle figure;

int choice;

double result;

cout << "Choose the figure type:\n";

cout << "1. Rectangle\n";

cout << "2. Square\n";

cout << "Enter your choice (1 or 2): ";

cin >> choice;

if (choice == 1) {

double length, width;

cout << "Enter the length of the rectangle: ";

cin >> length;

cout << "Enter the width of the rectangle: ";

cin >> width;

result = figure.rect\_area(length, width);

cout << "Area of the rectangle: " << result << endl;

}

else if (choice == 2) {

double side;

cout << "Enter the side of the square: ";

cin >> side;

result = figure.rect\_area(side);

cout << "Area of the square: " << result << endl;

}

else {

cout << "Invalid choice. Please choose 1 or 2.\n";

}

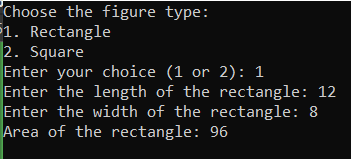
return 0;

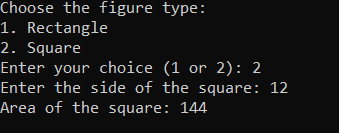
}

Программа представляет собой пример класса Rectangle, в котором перегружен метод rect\_area() для вычисления площади прямоугольника или квадрата в зависимости от выбора пользователя. Пользователю предлагается выбрать тип фигуры (прямоугольник или квадрат) и ввести соответствующие параметры. Затем программа использует соответствующую версию метода rect\_area() для расчета площади выбранной фигуры и выводит результат.

Программа демонстрирует использование перегрузки метода в классе для поддержки различных вариантов использования в зависимости от контекста.

**Работа программы:**





**Вопросы:**  
1)Полиморфизм: Полиморфизм в объектно-ориентированном программировании - это способность объекта использовать методы базового класса таким образом, чтобы они были адаптированы под конкретный производный класс. Это может быть достигнуто через виртуальные функции и интерфейсы.

2)Перегрузка операторов: Перегрузка операторов - это возможность определения собственного поведения для стандартных операторов языка программирования в контексте пользовательских типов данных. Это делает код более читаемым и удобным, поддерживая привычный синтаксис.

3)Переопределение операторов: Переопределение операторов - это предоставление новой реализации для существующего оператора. В отличие от перегрузки, где добавляется поддержка для пользовательских типов, переопределение меняет поведение оператора для существующих типов.

4) Синтаксис операторной функции: Синтаксис операторной функции в C++ выглядит следующим образом:

cppCopy code

тип\_возвращаемого\_значения operator оператор(параметры) { // реализация оператора }

5) Операторы, которые можно перегружать, и те, которые нельзя: Можно перегружать многие бинарные и унарные операторы, такие как +, -, \*, /, ==, != и др. Нельзя перегружать операторы, такие как ::, .\*, ., ?:.

6) Зависимость между количеством аргументов в операторной функции и количеством операндов: Количество аргументов в операторной функции зависит от количества операндов, с которыми оператор работает. Например, бинарные операторы требуют два аргумента (левый и правый операнд), унарные - один аргумент.

7) Как перегруженные операции делают код более читаемым: Перегруженные операции позволяют использовать привычный синтаксис для пользовательских типов данных, делая код более читаемым и естественным. Например, перегрузка оператора + для класса может позволить использовать выражение obj1 + obj2 вместо вызова функции add(obj1, obj2).