**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

Урманбетов Султан

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**.

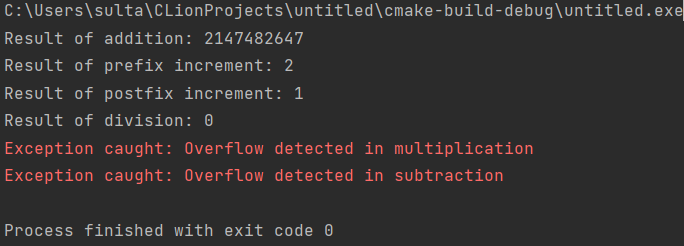
Если результат какой-либо из них выходит за границы типа **int** (в 32-битной системе), имеющее значения от **2 147 483 648** до **-2 147 483 648**, то операция должна послать сообщение об ошибке и завершить программу. Для выявления ошибки арифметического переполнения используйте концепцию **исключения**.

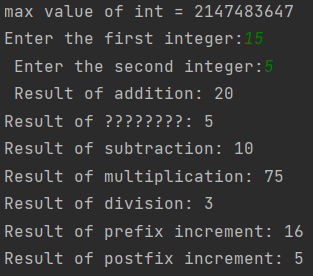
Для облегчения проверки переполнения выполняйте вычисления с использованием типа **long** **double**. При описании унарных операций используйте указатель **this**.

**Исходный код программы на C++:**

#include <iostream>  
#include <limits>  
  
using namespace std;  
  
class Int {  
private:  
 int value;  
  
public:  
 Int(int val = 0) : value(val) {}  
  
 // Overloaded arithmetic operators  
 Int operator+(const Int& other) const {  
 if ((long double)value + other.value > numeric\_limits<int>::max() ||  
 (long double)value + other.value < numeric\_limits<int>::min()) {  
 throw overflow\_error("Overflow detected in addition");  
 }  
 return Int(value + other.value);  
 }  
 Int operator+( int other) {  
 if ((long double)value - other > numeric\_limits<int>::max() ||  
 (long double)value - other < numeric\_limits<int>::min()) {  
 throw overflow\_error("Overflow detected in addition");  
 }  
 return Int(value - other - other);  
 }  
 Int operator-(const Int& other) const {  
 if ((long double)value - other.value > numeric\_limits<int>::max() ||  
 (long double)value - other.value < numeric\_limits<int>::min()) {  
 throw overflow\_error("Overflow detected in subtraction");  
 }  
 return Int(value - other.value);  
 }  
  
 Int operator\*(const Int& other) const {  
 if ((long double)value \* other.value > numeric\_limits<int>::max() ||  
 (long double)value \* other.value < numeric\_limits<int>::min()) {  
 throw overflow\_error("Overflow detected in multiplication");  
 }  
 return Int(value \* other.value);  
 }  
  
 Int operator/(const Int& other) const {  
 if (other.value == 0) {  
 throw invalid\_argument("Division by zero");  
 }  
 return Int(value / other.value);  
 }  
  
 // Prefix increment operator  
 Int& operator++() {  
 if (value == numeric\_limits<int>::max()) {  
 throw overflow\_error("Overflow detected in prefix increment");  
 }  
 ++value;  
 return \*this;  
 }  
  
 // Postfix increment operator  
 Int operator++(int) {  
 if (value == numeric\_limits<int>::max()) {  
 throw overflow\_error("Overflow detected in postfix increment");  
 }  
 Int temp(\*this);  
 ++value;  
 return temp;  
 }  
  
 // Output operator  
 friend ostream& operator<<(ostream& os, const Int& obj) {  
 return os << obj.value;  
 }  
  
 // Input operator  
 friend istream& operator>>(istream& is, Int& obj) {  
 return is >> obj.value;  
 }  
};  
  
int main() {  
 Int a, b;  
 cout<<"max value of int = 2147483647"<<endl;  
 cout << "Enter the first integer: ";  
 cin >> a;  
  
 cout << "Enter the second integer: ";  
 cin >> b;  
  
 try {  
 Int c = a + b;  
 cout << "Result of addition: " << c << endl;  
 } catch (const exception& e) {  
 cerr << "Exception caught: " << e.what() << endl;  
 }  
 try {  
 Int c = a + 5;  
 cout << "Result of ????????: " << c << endl;  
 } catch (const exception& e) {  
 cerr << "Exception caught: " << e.what() << endl;  
 }  
  
 try {  
 Int d = a - b;  
 cout << "Result of subtraction: " << d << endl;  
 } catch (const exception& e) {  
 cerr << "Exception caught: " << e.what() << endl;  
 }  
  
 try {  
 Int e = a \* b;  
 cout << "Result of multiplication: " << e << endl;  
 } catch (const exception& e) {  
 cerr << "Exception caught: " << e.what() << endl;  
 }  
  
 try {  
 Int f = a / b;  
 cout << "Result of division: " << f << endl;  
 } catch (const exception& e) {  
 cerr << "Exception caught: " << e.what() << endl;  
 }  
  
 try {  
 Int g = ++a;  
 cout << "Result of prefix increment: " << g << endl;  
 } catch (const exception& e) {  
 cerr << "Exception caught: " << e.what() << endl;  
 }  
  
 try {  
 Int h = b++;  
 cout << "Result of postfix increment: " << h << endl;  
 } catch (const exception& e) {  
 cerr << "Exception caught: " << e.what() << endl;  
 }  
  
 return 0;  
}

**Тест 1:**

****

**Тест 2:  
**

# **Задание №2**

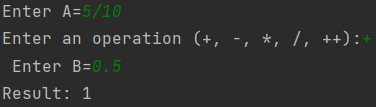
**Постановка задачи:**

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.

**Исходный код программы на C++:**

#include <iostream>  
#include <limits>  
  
using namespace std;  
  
class Int {  
private:  
 int value;  
  
public:  
 Int(int val = 0) : value(val) {}  
  
 // Overloaded arithmetic operators  
 Int operator+(const Int& other) const {  
 if ((long double)value + other.value > numeric\_limits<int>::max() ||  
 (long double)value + other.value < numeric\_limits<int>::min()) {  
 throw overflow\_error("Overflow detected in addition");  
 }  
 return Int(value + other.value);  
 }  
 Int operator+( int other) {  
 if ((long double)value - other > numeric\_limits<int>::max() ||  
 (long double)value - other < numeric\_limits<int>::min()) {  
 throw overflow\_error("Overflow detected in addition");  
 }  
 return Int(value - other - other);  
 }  
 Int operator-(const Int& other) const {  
 if ((long double)value - other.value > numeric\_limits<int>::max() ||  
 (long double)value - other.value < numeric\_limits<int>::min()) {  
 throw overflow\_error("Overflow detected in subtraction");  
 }  
 return Int(value - other.value);  
 }  
  
 Int operator\*(const Int& other) const {  
 if ((long double)value \* other.value > numeric\_limits<int>::max() ||  
 (long double)value \* other.value < numeric\_limits<int>::min()) {  
 throw overflow\_error("Overflow detected in multiplication");  
 }  
 return Int(value \* other.value);  
 }  
  
 Int operator/(const Int& other) const {  
 if (other.value == 0) {  
 throw invalid\_argument("Division by zero");  
 }  
 return Int(value / other.value);  
 }  
  
 // Prefix increment operator  
 Int& operator++() {  
 if (value == numeric\_limits<int>::max()) {  
 throw overflow\_error("Overflow detected in prefix increment");  
 }  
 ++value;  
 return \*this;  
 }  
  
 // Postfix increment operator  
 Int operator++(int) {  
 if (value == numeric\_limits<int>::max()) {  
 throw overflow\_error("Overflow detected in postfix increment");  
 }  
 Int temp(\*this);  
 ++value;  
 return temp;  
 }  
  
 // Output operator  
 friend ostream& operator<<(ostream& os, const Int& obj) {  
 return os << obj.value;  
 }  
  
 // Input operator  
 friend istream& operator>>(istream& is, Int& obj) {  
 return is >> obj.value;  
 }  
};  
  
int main() {  
 Int a, b;  
 cout<<"max value of int = 2147483647"<<endl;  
 cout << "Enter the first integer: ";  
 cin >> a;  
  
 cout << "Enter the second integer: ";  
 cin >> b;  
  
 try {  
 Int c = a + b;  
 cout << "Result of addition: " << c << endl;  
 } catch (const exception& e) {  
 cerr << "Exception caught: " << e.what() << endl;  
 }  
 try {  
 Int c = a + 5;  
 cout << "Result of ????????: " << c << endl;  
 } catch (const exception& e) {  
 cerr << "Exception caught: " << e.what() << endl;  
 }  
  
 try {  
 Int d = a - b;  
 cout << "Result of subtraction: " << d << endl;  
 } catch (const exception& e) {  
 cerr << "Exception caught: " << e.what() << endl;  
 }  
  
 try {  
 Int e = a \* b;  
 cout << "Result of multiplication: " << e << endl;  
 } catch (const exception& e) {  
 cerr << "Exception caught: " << e.what() << endl;  
 }  
  
 try {  
 Int f = a / b;  
 cout << "Result of division: " << f << endl;  
 } catch (const exception& e) {  
 cerr << "Exception caught: " << e.what() << endl;  
 }  
  
 try {  
 Int g = ++a;  
 cout << "Result of prefix increment: " << g << endl;  
 } catch (const exception& e) {  
 cerr << "Exception caught: " << e.what() << endl;  
 }  
  
 try {  
 Int h = b++;  
 cout << "Result of postfix increment: " << h << endl;  
 } catch (const exception& e) {  
 cerr << "Exception caught: " << e.what() << endl;  
 }  
  
 return 0;  
}

# **Тест:**



# **Задание №3**

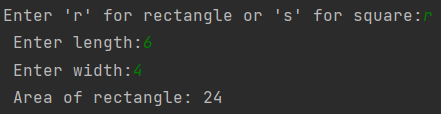
**Постановка задачи:**

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

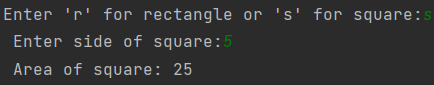
Этот метод возвращает площадь прямоугольника. В этой программе метод **rect\_area()** перегружается двумя способами. В первом — методу передаются оба размера фигуры. Эта версия используется для прямоугольника. Однако, в случае квадрата необходимо задавать только один аргумент, поэтому вызывается вторая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

**Исходный код программы на C++:**

# **Тест 1: прямоугольник**



# **Тест 2: квадрат**

****

# **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?**

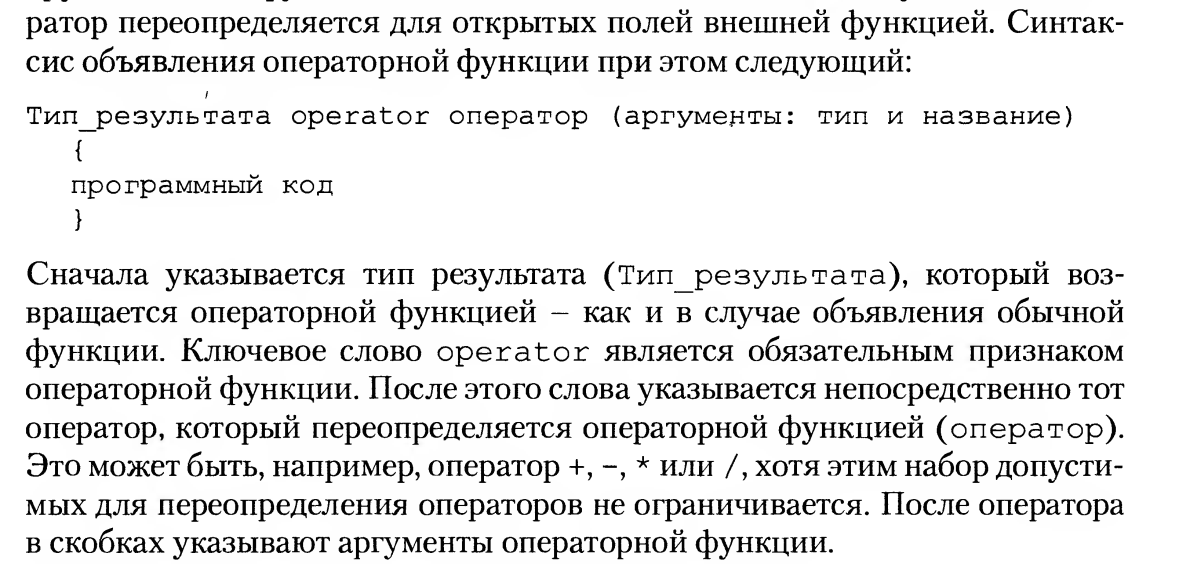
**Перегрузка операторов** – это возможность заставить компилятор осуществлять стандартные операции над нестандартными (пользовательскими) типами данных.

**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;