

# Перегрузка операторов

# Задача

Реализовать тип Rational для представления простых дробей.

$$9\frac{3}{4}$$

## Попытка 1: C-style

```
struct Rational {  
    int numerator_;  
    int denominator_;  
};
```

*Покритикуйте этот подход*

```
Rational rational_add(Rational lhs, Rational rhs);
```

```
Rational rational_sub(Rational lhs, Rational rhs);
```

## Проблема 1: нарушение инварианта

```
Rational r{3, 4};
```

```
r.denominator_ = 0;
```

## Проблема 2: Использование в обобщенном коде

```
const std::vector<Rational> rationals{  
    Rational{1, 2}, Rational{1, 4}, Rational{2, 8}};
```

**// Указатель на функцию**

```
const auto sum = std::accumulate(  
    rationals.begin(), rationals.end(), Rational{0, 1},  
    rational_add);
```

## Проблема 2: Использование в обобщенном коде

```
const std::vector<Rational> rationals{  
    Rational{1, 2}, Rational{1, 4}, Rational{2, 8}};
```

**// Громоздкая лямбда**

```
const auto sum = std::accumulate(  
    rationals.begin(), rationals.end(), Rational{0, 1},  
    [](auto lhs, auto rhs) {  
        return rational_add(lhs, rhs);  
    });
```

# Проблемы

1. Вызов функции по указателю нагружает CPU
2. Выписывание больших лямбд нагружает разработчика

Мы хотели бы, чтобы тип вел себя естественно и мог использовать разумные умолчания. Пробуйте сравнивать свои типы со встроенными (int).

## Попытка 2: C-style сокрытие данных

```
// Rational.hpp
```

```
struct Rational;
```

```
Rational* rational_new(  
    int numerator,  
    int denominator);
```

```
// Rational.cpp
```

```
struct Rational {  
    int numerator_, denominator_;  
};
```

```
Rational* rational_new(int n,  
                        int d) {  
    if (d == 0)  
        return nullptr;  
    // ...  
}
```



# Недостатки C-Style решения

1. Динамическое выделение памяти для экземпляров Rational.
2. Rational\* сохраняет интерфейс указателя:

```
auto* rp = make_rational(3, 4);
```

```
// Разрешена адресная арифметика.
```

```
auto* rpinv = rp + 1;
```

```
// Разрешено разыменование.
```

```
auto& r = *rp
```

```
// В множестве два элемента
```

```
std::set s{make_rational(3, 4), make_rational(3, 4)};
```

Мы хотели бы ограничить интерфейс.

## Попытка 3: класс

```
class Rational {  
    public:  
        Rational(int n, int d)  
            : numerator_(n),  
              denominator_(d) {}  
  
        int numerator();  
        int denominator();  
  
    private: ...  
};
```

*Покритикуйте это решение*

## Попытка 3: класс

```
class Rational {  
    public:  
        Rational(int n, int d)  
            : numerator_(n),  
              denominator_(d) {}  
  
        int numerator() const;  
        int denominator() const;  
  
    private: ...  
};
```

*Геттеры не меняют  
состояние экземпляра*

## Попытка 3: класс

```
class Rational {  
    public:  
        Rational(int n, int d)  
            : numerator_(n),  
              denominator_(d) {  
            if (denominator_ == 0) {  
                throw ...;  
            }  
            normalize();  
        }  
};
```

*Конструктор должен  
устанавливать инвариант*

## Попытка 3: класс

```
class Rational {  
    public:  
        Rational(int n, int d);  
};
```

*Удобно ли пользоваться таким  
конструктором?*

## Попытка 3: класс

```
class Rational {  
    public:  
        Rational(int n = 0, int d = 1);  
};
```

*Может ли конструктор быть вызван с одним аргументом?  
Что из этого следует?*

## explicit конструктор

```
void f(Rational r);
```

```
explicit Rational(int n = 0, int d = 1);
```

```
// conversion from 'int' to non-scalar type 'Rational'
```

```
Rational r = 1; // fix: Rational r(1);
```

```
// could not convert '1' from 'int' to 'Rational'
```

```
f(1); // fix: f(Rational(1))
```

## explicit(false) конструктор

```
void f(Rational r);
```

```
Rational(int n = 0, int d = 1);
```

```
Rational r = 1;    // ok
```

```
f(1);              // ok
```



# Реализуем арифметические операции

```
class Rational {  
    public:  
        // Вариант 1: метод класса  
        Rational operator+(Rational rhs) const;  
};  
  
// Вариант 2: свободная функция  
Rational operator+(Rational lhs, Rational rhs);
```

## Метод класса нарушает коммутативность

```
const Rational ra(1, 4);
```

```
const auto rd = ra + 1; // ra.operator+(1)
```

```
const auto re = 1 + ra; // 1.operator+(ra);
```

## Коммутативность сложения сохраняется

```
Rational operator+(Rational lhs, Rational rhs);
```

```
const Rational ra(1, 4);
```

```
const auto rd = ra + 1; // operator+(ra, 1)
```

```
const auto re = 1 + ra; // operator+(1, ra)
```

Обычно типы располагаются в неймспейсе

```
namespace mymath {  
  
class Rational { ... };  
Rational operator+(Rational lhs, Rational rhs);  
  
}  
  
const mymath::Rational ra(1, 4);  
  
// Нет синтаксиса для указания пространства имен  
const auto rd = ra + 1;
```

# ADL: Argument-dependent lookup

Argument-dependent lookup, also known as ADL, or Koenig lookup, is the set of rules for looking up the unqualified function names in function-call expressions, including implicit function calls to overloaded operators. These function names are looked up in the namespaces of their arguments in addition to the scopes and namespaces considered by the usual unqualified name lookup.

## Оператор +=

```
class Rational {  
    public:  
        // 1  
        Rational& operator+=(Rational rhs);  
};
```

```
// 2  
Rational& operator+=(Rational& lhs, Rational rhs);
```

## Оператор +=

**// Не сработает. Объясните, почему.**

```
1 += Rational(2, 1);
```

## Реализация + через +=

```
Rational operator+(Rational lhs, Rational rhs) {  
    lhs += rhs;  
    return lhs;  
}
```



## Переход к шаблону

```
template <typename T>
class Rational {
private:
    T numerator_, denominator_;
};
```

```
template <typename T>
Rational<T> operator-(Rational<T> lhs,
                     Rational<T> rhs);
```

*Как изменится  
сигнатура оператора?*

## Переход к шаблону

```
template <typename T>
class Rational {
    private:
        T numerator_, denominator_;
};
```

*Размер экземпляра  
теперь неизвестен*

```
template <typename T>
Rational<T> operator-(const Rational<T>& lhs,
                      const Rational<T>& rhs);
```

## Переход к шаблону

```
template <typename T>
class Rational {
private:
    T numerator_, denominator_;
};
```

*Можно ли использовать  
указатели?*

```
template <typename T>
Rational<T> operator-(const Rational<T>* lhs,
                     const Rational<T>* rhs);
```

## Переход к шаблону

```
template <typename T>
class Rational {
private:
    T numerator_, denominator_;
};
```

*operator- уже определен  
для указателей*

```
template <typename T>
Rational<T> operator-(const Rational<T>* lhs,
                     const Rational<T>* rhs);
```

# За кадром

Еще есть:

- Бинарные  $+$   $-$   $*$   $/$   $\%$  и парные им  $+=$  ...
- Унарные  $+$   $-$
- Битовые  $|$   $\&$   $^$   $\sim$   $<<$   $>>$

Мы не будем детально рассматривать каждый оператор.

# Перегрузка инкремента

```
class Rational {
```

```
public:
```

```
...
```

```
Rational operator++(int);
```

```
Rational& operator++();
```

```
...
```

```
};
```

*Префиксный и постфиксный инкремент.*

*– Где какой?*

*– Зачем нужен фиктивный аргумент?*

# Перегрузка инкремента

```
class Rational {  
    public:  
        ...  
        Rational operator++(int);  
        Rational& operator++();  
        ...  
};
```

```
int x = 42;  
auto a = ++x;    // ?  
auto b = x++;    // ?  
                // x = ?
```

## Перегрузка инкремента

```
class Rational {  
    public:  
        ...  
        Rational operator++(int);  
        Rational& operator++();  
        ...  
};
```

```
int x = 42;  
auto a = ++x;    // 43  
auto b = x++;    // 43  
// x = 44
```



## Реализация x++ через ++x

```
Rational operator++(int) {  
    const auto old = *this;  
    ++(*this);  
    return old;  
}
```

# Оператор вывода в поток

```
#include <ios_fwd>
```

```
std::ostream& operator<<(std::ostream& os,  
                        Rational r);
```

Может быть только внешней функцией.

Допустим, `operator<<` — метод...

```
class Rational {  
    public:  
        std::ostream& operator<<(std::ostream& os) const;  
};
```

Тогда вызов:

```
r << std::cout;
```

# Операторы преобразования типа

```
class Rational {  
    public:  
        ...  
        operator double() const;  
        ...  
};
```

# Контроль за неявными преобразованиями

```
explicit operator double() const;
```

```
double x = r; // cannot convert from 'Rational' to 'double'
```

```
auto x = static_cast<double>(r); // Ok
```

```
auto y = r.operator double(); // Ok, но не пишите так
```

```
/* explicit(false) */ operator double() const;
```

```
double x = r; // Ok
```

# Литералы

```
auto x1 = 42;      // int
auto x2 = 42.0;    // double
auto x3 = 42.0f;   // float
auto x4 = 42ul;    // unsigned long
```

```
using namespace std::string_literals;
using namespace std::string_view_literals;
```

```
auto x5 = "42"s;   // std::string
auto x6 = "42"sv;  // std::string_view
```

# Пользовательские литералы

```
namespace literals {  
  
Rational operator"" _r(unsigned long long value) {  
    return Rational(static_cast<int>(value));  
}  
  
}
```

## Пример использования

```
using namespace csc::literals;
```

```
auto r1 = 3_r;      // 3/1
```

```
auto r2 = 3 / 4_r;  // 3/4
```



# CppCoreGuidelines

## [C.over: Overloading and overloaded operators](#)

C.160: Define operators primarily to mimic conventional usage

C.161: Use non-member functions for symmetric operators

C.167: Use an operator for an operation with its conventional meaning

C.168: Define overloaded operators in the namespace of their operands

# CppCoreGuidelines

## [C.over: Overloading and overloaded operators](#)

C.160: Define operators primarily to mimic conventional usage

C.161: Use non-member functions for symmetric operators

**C.167: Use an operator for an operation with its conventional meaning**

C.168: Define overloaded operators in the namespace of their operands

# Conventional meaning

- Сравнение (`==`, `!=`, `<`, `<=`, `>`, `>=`, `<=>`)
- Арифметические операции (`+`, `-`, `*`, `/`, `%`)
- Аксессоры (`->`, unary `*`, `[]`)
- Присваивание (`=`)

Не надо так

```
// ~my_string == my_string.size()  
for (std::size_t i = 0; i < ~my_string; ++i) ...
```

```
BigInteger x, y;  
auto res = x ^ y; // Возведение в степень
```

*Чем плоха такая перегрузка?*

# The Functional Template Library: Монады

```
shared_ptr<Widget> foo();  
shared_ptr<Gadget> bar(Widget w);  
  
auto ptr = foo() >>= bar;  
  
shared_ptr<Gadget> ptr(nullptr);  
auto ptrw = foo();  
if (ptrw) {  
    ptr = bar(*ptrw);  
}
```

## Стало идиоматичным

```
std::string a = "aaa", b = "bbb";
```

**// Нарушается коммутативность сложения**

```
auto res = a + b;
```

## Стало идиоматичным

```
std::filesystem::path home_dir = "/home/username";  
std::filesystem::path readme_file_name = "readme.md";
```

// Мимикрирует под формат

```
auto readme_path = home_dir / readme_file_name;
```

## Стало идиоматичным

**// Многострадальные операторы << и >>**  
`std::cout << readme_path << '\n';`



# Логические операторы

Что будет выведено?

```
bool f() { std::cout << "f()\n"; return true; }  
bool g() { std::cout << "g()\n"; return true; }  
  
if (f() || g()) {  
    std::cout << "Ok\n";  
}
```

# Логические операторы

Пусть дано выражение:

`f() || f()`

- Если оператор `||` встроенный и левое выражение вернет `true`, то правое не будет вычисляться
- Если оператор `||` перегружен пользователем, то оба операнда вычисляются энергично:

**`operator ||(f(), f());`**

Теряется ленивость операторов.

# Ремонт нельзя завершить

```
template <class T>
BOOST_CXX14_CONSTEXPR
typename boost::enable_if_c<
    rational_detail::is_compatible_integer<T, IntType>::value,
    rational&>::type
operator+=(const T& i);
```

# Конструктор

```
template <class T, class U>
BOOST_CXX14_CONSTEXPR rational(const T& n, const U& d, typename enable_if_c<
    (!rational_detail::is_compatible_integer<T, IntType>::value
    || !rational_detail::is_compatible_integer<U, IntType>::value)
    && std::numeric_limits<T>::is_specialized &&
    std::numeric_limits<T>::is_integer
    && (std::numeric_limits<T>::radix == std::numeric_limits<IntType>::radix)
    && is_convertible<T, IntType>::value &&
    std::numeric_limits<U>::is_specialized && std::numeric_limits<U>::is_integer
    && (std::numeric_limits<U>::radix == std::numeric_limits<IntType>::radix)
    && is_convertible<U, IntType>::value
>::type const* = 0)
{ assign(n, d); }
```

# Вывод

В C++ среди прочих инструментов для проектирования типов есть:

1. Неявные преобразования (с помощью конструктора и операторов)
2. Перегрузка операторов
3. Пользовательские литералы

Это позволяет разработчику создавать типы, которые ведут себя практически как встроенные.

Вопросы?