# Перегрузка операторов

# TL;DR Перегрузка операторов

Пользовательские типы ничем не хуже встроенных.

```
struct Rational { int num_, denom_; };
Rational operator+(Rational lhs, Rational rhs);
Rational operator-(Rational lhs, Rational rhs);
…
Rational a{1, 4};
auto b = a + a; // b == {1, 2}
```

# Задача

Реализовать тип Rational для представления простых дробей.

# Попытка 1: C-style

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

Rational rational_add(Rational lhs, Rational rhs);

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

*Покритикуйте этот подход*

# Проблема 1: нарушение инварианта

```
Rational r{3, 4};

r.denominator_ = 0;
```

# Проблема 2: Использование в обобщенном коде

```
const std::vector 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 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[1]
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).

[1] В предыдущем примере все равно можем упереться в вызов rational_add. Универсальная рекомендация: производительность нужно замерять.

# 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 ctor 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))
```



https://godbolt.org/z/GxGfnjba8

# explicit(false) конструктор

```
void f(Rational r);

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

Rational r = 1;  // Ok
f(1);            // Ok
```



https://godbolt.org/z/Mjnee6ovo

# 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);
```



https://godbolt.org/z/MT4GbEo4K

# 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)
```



https://godbolt.org/z/TYehsfrfo

# 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.



Argument-dependent 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_;
};

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

*Размер экземпляра теперь неизвестен*

# Переход к шаблону

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

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

*Можно ли использовать указатели?*

# Переход к шаблону

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

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

operator– уже определен для указателей

# За кадром

Еще есть:

- Бинарные + − * / % и парные им += …
- Унарные + −
- Битовые | & ^ ~ << >>

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 в вызове операнды будут перепутаны местами,
это выглядит как чтение из потока вывода:

```
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 (ptra) {

  ptr = bar(*ptrw);

}
```



FTL: Expanding The Standard Library

# 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’;
```

# 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.

TODO: не забыть про докладчиков

# Q&A