# В предыдущей серии

# Основные тезисы

- Пользовательские типы не хуже встроенных
- Средства:
  - Управление строгостью типизации (explicit ctor & cast)
  - Перегрузка операторов
- Разобрали:
  - Бинарные арифметические операторы на примере +
  - Префиксные и постфиксные ++ и −−
  - Нельзя переопределять операторы для встроенных типов

# What if…?

```
struct Integer {
  Integer operator++(int x);
  int value_ = 0;
};

Integer x{2};
x.operator++(40);
```

[over.inc]



https://godbolt.org/z/1PWq9chMv

# Разминка: что такое f?

```
f(x); // вызов f от аргумента x
```

# Разминка

```
f(x); // вызов f от аргумента x
```

- Функция (или указатель или ссылка на функцию)
- Экземпляр класса с перегруженным `operator()`
- Объект, неявно приводимый к чему-либо выше
- Шаблон
- Специализация шаблона
- Макрос
- Тип (выражение — вызов конструктора)

# Операторы сравнения

# Пусть дан класс

```
struct Int {
  Int(int v): v_(v) {}
  int v_;
};
```

- Пока нам неважно сокрытие данных
- Пример еще проще, чем Rational

# Операторы сравнения в C++17

```
bool operator< (Int lhs, Int rhs) { return lhs.v_ < rhs.v_; }
bool operator<=(Int lhs, Int rhs) { return !(rhs < lhs); }
bool operator> (Int lhs, Int rhs) { return rhs < lhs; }
bool operator>=(Int lhs, Int rhs) { return !(lhs < rhs); }
bool operator==(Int lhs, Int rhs) { return lhs.v_ == rhs.v_; }
bool operator!=(Int lhs, Int rhs) { return !(lhs == rhs); }
```

# Операторы сравнения в C++17

```
bool operator< (Int lhs, Int rhs) { return lhs.v_ < rhs.v_; }
bool operator<=(Int lhs, Int rhs) { return !(rhs < lhs); }
bool operator> (Int lhs, Int rhs) { return rhs < lhs; }
bool operator>=(Int lhs, Int rhs) { return !(lhs < rhs); }
bool operator==(Int lhs, Int rhs) { return lhs.v_ == rhs.v_; }
bool operator!=(Int lhs, Int rhs) { return !(lhs == rhs); }
```

*Почему внешние функции, а не методы класса?*

# Операторы сравнения в C++17

```
bool operator< (Int lhs, Int rhs) { return lhs.v_ < rhs.v_; }
bool operator<=(Int lhs, Int rhs) { return !(rhs < lhs); }
bool operator> (Int lhs, Int rhs) { return rhs < lhs; }
bool operator>=(Int lhs, Int rhs) { return !(lhs < rhs); }
bool operator==(Int lhs, Int rhs) { return lhs.v_ == rhs.v_; }
bool operator!=(Int lhs, Int rhs) { return !(lhs == rhs); }
```

*Два основных оператора: < и ==*

# Проблемы операторов сравнения в C++17

1. Очевидно: Основные операторы < и ==,
   остальные выражаются через них.
   Мы хотели бы сократить количество кода.
2. Менее очевидно: на 6 строк кода 12 ошибок.
3. Совсем неочевидно: в некоторых случаях 6 операторов мало.

# Проблемы операторов сравнения в C++17

1. **Очевидно: Основные операторы < и ==,**
   **остальные выражаются через них.**
   **Мы хотели бы сократить количество кода.**
2. Менее очевидно: на 6 строк кода 12 ошибок.
3. Совсем неочевидно: в некоторых случаях 6 операторов мало.

# std::rel_ops (deprecated in C++20)

```
// You write
using namespace std::rel_ops;

// You get
template <typename T>
bool operator!=(const T& lhs, const T& rhs);
              >
              <=
              >=
```



std::rel_ops

# Проблемы операторов сравнения в C++17

1. Очевидно: Основные операторы < и ==,
   остальные выражаются через них.
   Мы хотели бы сократить количество кода.
2. **Менее очевидно: на 6 строк кода 12 ошибок.**
3. Совсем неочевидно: в некоторых случаях 6 операторов мало.

# Операторы должны быть constexpr и noexcept

```
constexpr bool operator< (Int lhs, Int rhs) noexcept { ... }
constexpr bool operator<=(Int lhs, Int rhs) noexcept { ... }
constexpr bool operator> (Int lhs, Int rhs) noexcept { ... }
constexpr bool operator>=(Int lhs, Int rhs) noexcept { ... }
constexpr bool operator==(Int lhs, Int rhs) noexcept { ... }
constexpr bool operator!=(Int lhs, Int rhs) noexcept { ... }
```

*Конструктор Int тоже должен быть constexpr*

# Проблемы операторов сравнения в C++17

1. Очевидно: Основные операторы < и ==,
   остальные выражаются через них.
   Мы хотели бы сократить количество кода.
2. Менее очевидно: на 6 строк кода 12 ошибок.
3. **Совсем неочевидно: в некоторых случаях 6 операторов мало.**

# Задача

- Вы разрабатываете класс String
- Сколько операторов сравнения нужно перегрузить и почему?

# Считаем: 6 операторов по шаблону

```
bool operator< (const String& lhs, const String& rhs);
bool operator<=(const String& lhs, const String& rhs);
bool operator> (const String& lhs, const String& rhs);
bool operator>=(const String& lhs, const String& rhs);
bool operator==(const String& lhs, const String& rhs);
bool operator!=(const String& lhs, const String& rhs);
```

# …еще 12 для сравнения с const char*

```
bool operator< (const char* lhs, const String& rhs);
bool operator< (const String& lhs, const char* rhs);
...
```

# Это слишком много!

```
bool operator< (const String& lhs, const String& rhs);
bool operator<=(const String& lhs, const String& rhs);
bool operator> (const String& lhs, const String& rhs);
bool operator>=(const String& lhs, const String& rhs);
bool operator==(const String& lhs, const String& rhs);
bool operator!=(const String& lhs, const String& rhs);

bool operator< (const char* lhs, const String& rhs);
bool operator<=(const char* lhs, const String& rhs);
bool operator> (const char* lhs, const String& rhs);
bool operator>=(const char* lhs, const String& rhs);
bool operator==(const char* lhs, const String& rhs);
bool operator!=(const char* lhs, const String& rhs);

bool operator< (const String& lhs, const char* rhs);
bool operator<=(const String& lhs, const char* rhs);
bool operator> (const String& lhs, const char* rhs);
bool operator>=(const String& lhs, const char* rhs);
bool operator==(const String& lhs, const char* rhs);
bool operator!=(const String& lhs, const char* rhs);
```

*Зачем это все?*

# Конструктор String может выделять память

Мы не хотим аллокаций из-за неявных преобразований:

```
class String {
 public:
  String(const char* str):
    len_(strlen(str)),
    buf_(new char[len_]) { ... }
};
```

*В реальности все несколько сложнее*

# Решение C++20

1. Three-way comparison operator <=>.
2. Механика переписывания (rewrite) и обращения (reverse) операторов.

# Three-way comparison operator

Формула: `a @ b -> (a <=> b) @ 0`

# Three-way comparison operator

Формула: `a @ b -> (a <=> b) @ 0`

```
bool a = (3 <=> 4) < 0;  // 3 < 4, true
bool b = (4 <=> 4) == 0; // 4 == 4, true
bool c = (4 <=> 3) > 0;  // 4 > 3, true
```



https://godbolt.org/z/TcrE8nfK8

# Three-way comparison operator

Формула: `a @ b -> (a <=> b) @ 0`

```
bool a = (3 <=> 4) < 0;  // 3 < 4, true
bool b = (4 <=> 4) == 0; // 4 == 4, true
bool c = (4 <=> 3) > 0;  // 4 > 3, true
```

*Вам это что-нибудь напоминает?*



https://godbolt.org/z/TcrE8nfK8

# Three-way comparison operator

Формула: `a @ b -> (a <=> b) @ 0`

```
bool a = (3 <=> 4) < 0;  // 3 < 4, true
bool b = (4 <=> 4) == 0; // 4 == 4, true
bool c = (4 <=> 3) > 0;  // 4 > 3, true
```

*Что возвращает operator<=>?*



https://godbolt.org/z/TcrE8nfK8

# operator<=>

```
class Widget {
 public:
  /* comp_cat */ operator<=>(const Widget& rhs);

  // One of:
  std::strong_ordering operator<=>(const Widget& rhs);
  std::weak_ordering operator<=>(const Widget& rhs);
  std::partial_ordering operator<=>(const Widget& rhs);
};
```

# std::strong_ordering

- Похоже на сравнение int’ов
- Равенство подразумевает взаимозаменяемость:
  Если a == b, то f(a) == f(b)

Допустимые значения:

- std::strong_ordering::less
- std::strong_ordering::equal (+ std::strong_ordering::equivalent)
- std::strong_ordering::greater

# std::weak_ordering

- Похоже на регистронезависимое сравнение строк
- Равенство определяет класс эквивалентности:
  “hello” == “HELLO”
- Строки эквивалентны, но не равны, поэтому
  f(“hello”) ?? f(“HELLO”)

Допустимые значения:

- std::weak_ordering::less
- std::weak_ordering::equivalent
- std::weak_ordering::greater

# std::partial_ordering

- Похоже на сравнение floating point
- std::weak_ordering::{less, equivalent, greater}
- std::weak_ordering::unordered

# std::partial_ordering

```
const auto qnan =
std::numeric_limits<double>::quiet_NaN();

const bool a = qnan == qnan;  // false
const bool b = qnan < qnan;   // false
const bool c = qnan > qnan;   // false
```



https://godbolt.org/z/j4vGqWPY3

# std::partial_ordering::unordered

```
// true
(qnan <=> qnan) == std::partial_ordering::unordered;
```



https://godbolt.org/z/j4vGqWPY3

# Пример

```
struct Rect {
  std::weak_ordering operator<=>(Rect rhs) const {
    return area() <=> rhs.area();
  }

  int area() const { return width_ * height_; }

  int width_ = 0;
  int height_ = 0;
};
```



https://godbolt.org/z/1rTs95c34

# Операторы сравнения в C++20

Primary: ==, <=>

Secondary: !=, <, >, <=, >=

Primary могут быть обращены

Secondary могут быть переписаны

# Пример: обращение ==

```
struct Integer {
  Int(int v) : v_(v) {}

  bool operator==(Int rhs) const {
    return v_ == rhs.v_;
  }

  int v_ = 0;
};
```

*Метод, не внешняя функция!*

# Пример: обращение ==

```
Integer a = 1;

// C++17
bool a_eq_b = a == 1;  // Ok: a.operator==(b)
bool b_eq_a = 1 == a;  // Error: 1.operator==(a)
```

# Пример: обращение ==

```
Integer a = 1;

// C++20
bool a_eq_b = a == 1;  // Ok: a.operator==(b)
bool b_eq_a = 1 == a;  // Error: 1.operator==(a)
                       // Rewrite: a.operator==(1)
```

# Пример: переписывание !=

```
Integer a = 1;

// C++17
bool a_neq_b = a != 1;  // Error
bool b_neq_a = 1 != a;  // Error
                        // no match for 'operator!='
```

# Пример: переписывание !=

```
Integer a = 1;

// C++20
bool a_neq_b = a != 1;  // Ok: a != 1 -> !(a == 1)

bool b_neq_a = 1 != a;  // Ok: 1 != a -> !(1 == a)
                                      -> !(a == 1)
```

# Возвращаемся к классу String

```
class String {
 public:
  std::strong_ordering
  operator<=>(const String& b) const;

  std::strong_ordering
  operator<=>(const char* b) const;
};
```

# Возвращаемся к классу String

```
class String {
 public:
  std::strong_ordering
  operator<=>(const String& b) const;

  std::strong_ordering
  operator<=>(const char* b) const;
};
```

*Кто видит проблему?*

# Возвращаемся к классу String

```
class String {
 public:
  bool operator==(const String& b) const;
  bool operator==(const char* b) const;

  std::strong_ordering
  operator<=>(const String& b) const;

  std::strong_ordering
  operator<=>(const char* b) const;
};
```

# Так себе пример

```
struct User {
  int id_;
  std::string first_name_;
  std::string last_name_;

  auto operator<=>(const User&) = default;
};
```

# Рекомендация

Перегружайте операторы сравнения для типов только если у них есть соответствующая семантика.

# Материалы

- Barry’s C++ Blog: Comparisons in C++20
- CppCon 2019: Jonathan Müller “Using C++20's Three-way Comparison <=>”

# Вопросы?