

# ОПЕРАТОРЫ

---

Перегрузка операторов. Бинарные операторы и цепочечные  
операторы. Экзотика и общий обзор чего не следует трогать.

К. Владимиров, Syntacore, 2023  
mail-to: konstantin.vladimirov@gmail.com

- Приведение типов
  - Операторы и цепочки
  - Бинарные операторы
  - Экзотика

# Типы гораздо важнее в C++ чем в C

- В заголовок этого слайда вынесено неоспоримое утверждение.
  - Типы участвуют в разрешении имён.
  - Типы могут иметь ассоциированное поведение.
  - За счёт шаблонной параметризации, типов может быть куда больше, их куда проще порождать из обобщённого кода.
- Но при всём этом, любой объект это просто кусок памяти.

```
float f = 1.0;
```

```
char x = *((char *)&f + 2); // Это легально. Что в x?
```

# Обсуждение

- Не имеет ли приведение в стиле C (реинтерпретация памяти) тёмных сторон?

# Обсуждение

- Не имеет ли приведение в стиле С (реинтерпретация памяти) тёмных сторон?
- Конечно имеет. Она слишком разрешающая. Есть некая разница между:
  - Приведением `int` к `double`.
  - Приведением `const int*` к `int*`.
  - Приведением `int*` к `long`.
- Первое это обычное дело, второе это опасное снятие внутренней константности, третье за гранью добра и зла.

`x = (T) y;`

- Но в языке С всё это пишется одинаково.

# Приведения в стиле C++

- `static_cast` – обычные безопасные преобразования.

```
int x;  
double y = 1.0;  
x = static_cast<int>(y);
```

- `const_cast` – снятие константности или волатильности.

```
const int *p = &x;  
int *q = const_cast<int*>(p);
```

- `reinterpret_cast` – слабоумие и отвага.

```
long long uq = *reinterpret_cast<long long*>(q);
```

# Приведения в стиле C++

- `static_cast` – обычные безопасные преобразования.
- `const_cast` – снятие константности или волатильности.
- `reinterpret_cast` – слабоумие и отвага, но лучше, чем C style cast.

```
char c;
std::cout << "char # " << static_cast<int>(c) << std::endl;

int i;
const int* p = &i;
std::cout << "int: " << *(const_cast<int*>(p)) << std::endl;
```

- В обоих этих случаях `reinterpret_cast` будет ошибкой компиляции.

# Избегаем reinterpret-cast в C++20

- Побитовая реинтерпретация значения очень коварна.

```
float p = 1.0;  
int n = *reinterpret_cast<int*>(&p); // [basic.lval/11] UB
```

- Чтобы вы так не делали, в C++ появилась функция std::bit\_cast.

```
int m = std::bit_cast<int>(p);
```

- Она делает примерно следующее:

```
std::memcpy(&m, &p, sizeof(int));
```

- И не вовлекает вас в грех перед строгим алиасингом.

# Functional style cast в C++

- Функциональный каст это C-style cast вывернутый наизнанку.

```
int a = (int) y; // C-style  
int b = int(y); // functional-style C-style cast
```

- Разницы между ними нет, но заметьте:

```
int c = int{y}; // ctor, блокирует сужающие преобразования  
int d = S(x, y); // ctor, два аргумента
```

- Неприятно иногда вместо честного конструирования вlipнуть в C-style cast.
- Итак почти всегда наш выбор этот static\_cast или нечто похожее.
- В частности он является нашим выбором для явных преобразований типов.

# Static cast это явное преобразование

- Уже рассмотренные нами explicit конструкторы регламентируют необходимость static\_cast.

```
struct T {};
struct S { explicit S(T) {} };
void foo(S s) {}
foo(T); // FAIL
foo(static_cast<S>(T)); // OK
```

- То же самое касается синтаксиса копирующей инициализации.

```
T x; S x = static_cast<S>(y); // OK
```

# Обсуждение

- Кроме того, что C++ style casts позволяют чётко указать что вы хотите, они ещё и лучше видны в коде.
- По ним проще искать, чтобы их удалить, потому что вообще-то в статически типизированном языке преобразование типов это сигнал о проблемах в проектировании.
- Самый "безопасный" static\_cast на самом деле самый сложный т.к. у него нет чётких правил что на входе и что на выходе.
- static\_cast определяет явные преобразования. Но как типы преобразуются неявными преобразованиями?

# Особенности неявного приведения

- В наследство от языка С нам достались неявные арифметические преобразования.

```
int a = 2; double b = 2.8;  
short c = a * b;           // c = ?
```

- Со своими странностями и засадами

```
unsigned short x = 0xFFFF, y = 0xEEEE; // x * y = 0xEEEC2224  
unsigned short v = x * y;           // v = ?  
unsigned w = x * y;               // w = ?  
unsigned long long z = x * y;     // z = ?
```

- Может ли кто-нибудь исчерпывающе изложить сишную часть правил?

# Особенности неявного приведения

- Синые правила (применять сверху вниз):

type `op` fptype => fptype `op` fptype

- Порядок: long double, double float.

type `op` unsigned itype => unsigned itype `op` unsigned itype

type `op` itype => itype `op` itype

- Порядок: long long, long, int.

(itype less then int) `op` (itype less then int) => int `op` int

- Любые комбинации (unsigned) short и (unsigned) char.

# Особенности неявного приведения

- Неявные касты на инициализации.

```
widetype x; narrowtype y;
```

```
[decayed] widetype z = y; // ok
```

```
[decayed] narrowtype v = x; // ok если v вмещает значение x
```

- Понятно что параметры функции это тоже инициализация.

```
void foo(double);
```

```
foo(5); // ok, int implicitly promoted
```

# Унарный плюс (positive hack)

- Оператор унарного плюса интересен тем, что для почти всех встроенных типов он не значит ничего. Например `2 == +2`.
- Но при этом он, [даже если не перегружен](#), предоставляет легальный способ вызвать приведение к встроенному типу.

```
struct Foo { operator long() { return 42; }};  
void foo(int x);  
void foo(Foo x);  
  
Foo f;  
foo(f); // вызовет foo(Foo)  
foo(+f); // вызовет foo(int)
```

- ❑ Приведение типов
- Операторы и цепочки
- ❑ Бинарные операторы
- ❑ Экзотика

# Ваши типы как встроенные

- Собственный класс кватернионов.

```
template<typename T> struct Quat {  
    T x, y, z, w;  
};
```

- У нас уже есть бесплатное копирование и присваивание. Хотелось бы чтобы работало всё остальное: сложение, умножение на число и так далее.
- Начнём с чего-нибудь простого.

```
Quat q {1, 2, 3, 4};
```

```
Quat p = -q; // унарный минус: {-1, -2, -3, -4}
```

# Общий синтаксис операторов

- Обычно используется запись operator и далее какой это оператор.

```
template<typename T> struct Quat {  
    T x, y, z, w;  
};  
  
template<typename T> Quat<T> operator-(Quat<T> arg) {  
    return Quat<T>{-arg.x, -arg.y, -arg.z, -arg.w};  
}
```

- Теперь всё как надо.

```
Quat q {1, 2, 3, 4};
```

```
Quat p = -q; // унарный минус: {-1, -2, -3, -4}
```

# Общий синтаксис операторов

- Альтернатива: метод в классе.

```
template<typename T> struct Quat {  
    T x, y, z, w;  
  
    Quat operator-() const {  
        return Quat{-x, -y, -z, -w};  
    }  
};
```

- И снова всё как надо.

```
Quat q {1, 2, 3, 4};
```

```
Quat p = -q; // унарный минус: {-1, -2, -3, -4}
```

# Обсуждение

- Обычно есть два варианта (исключение: присваивание и пара-тройка других).
- -а означает a.operator-()
- -а означает operator-(a)
- Как вы думаете, что будет если определить оба?

# Обсуждение

- Как вы думаете чем закончится попытка:
- перегрузить operator- для int.

```
int operator-(int x) {  
    std::cout << "MINUS!" << std::endl;  
    return x;  
}
```

- перегрузить operator- для всего подряд в том числе и для int.

```
template <typename T> T operator-(T x) {  
    std::cout << "MINUS!" << std::endl;  
    return x;  
}
```

# Обсуждение

- Унарный минус всё-таки немного сомнительный оператор для перегрузки.
- Давайте, прежде чем двигаться дальше, мотивируем перегрузку операторов, то есть покажем, как она даёт нам производительность и возможности.

# Функторы: постановка проблемы

- Эффективность `std::sort` резко проседает если для его объектов нет `operator<` и нужен кастомный предикат.

```
bool gtf(int x, int y) { return x > y; }

// неэффективно: вызовы по указателю
std::sort(myarr.begin(), myarr.end(), &gtf);
```

- Можно ли с этим что-то сделать?

# Функторы: первый вариант решения

- Функтором называется класс, который ведёт себя как функция.
- Простейший способ это неявное приведение к указателю на функцию.

```
struct gt {  
    static bool gtf(int x, int y) { return x > y; }  
    using gtfptr_t = bool (*)(int, int);  
    operator gtfptr_t() const { return gtf; }  
};  
  
// гораздо лучше: теперь возможна подстановка  
std::sort(myarr.begin(), myarr.end(), gt{});
```

- Увы, это жутковато выглядит и плохо расширяется.

# Функторы: перегрузка ()

- Более правильный способ сделать функтор это перегрузка вызова.

```
struct gt {  
    bool operator() (int x, int y) { return x > y; }  
};  
  
// всё так же хорошо  
std::sort(myarr.begin(), myarr.end(), gt{});
```

- Почти всегда это лучше, чем указатель на функцию.
- Кроме того в классе можно хранить состояние.
- Функторы с состоянием получат второе дыхание когда мы дойдём до так называемых [лямбда-функций](#).

# Идиома PImpl

- Идиома PImpl предполагает единичное владение.

```
class Ifacade {  
    CImpl *impl_;  
public:  
    Ifacade() : impl_(new CImpl) {}  
    // методы  
};
```

- Эта идиома очень полезна: в частности она позволяет всегда иметь объект класса одного и того же размера, что может быть очень важно в ABI.
- Хорошой ли идеей является здесь заменить `CImpl *` на `unique_ptr`?

# Проблема неполного типа

- Попробуем использовать unique pointers в PImpl.

```
class MyClass; // предварительное объявление

struct MyWrapper {
    MyClass *c; // это ок
    MyWrapper() : c(nullptr) {};
};

struct MySafeWrapper {
    unique_ptr<MyClass> c; // увы, не компилируется
    MySafeWrapper() : c(nullptr) {};
};
```

# Как реально выглядит unique\_ptr?

- Стратегия удаления у него вынесена в параметр шаблона.

```
template <typename T, typename Deleter = default_delete<T>>
class unique_ptr {
    T *ptr_; Deleter del_;
public:
    unique_ptr(T *ptr = nullptr, Deleter del = Deleter()) :
        ptr_(ptr), del_(del) {}
    ~unique_ptr() { del_(ptr_); }
    // и так далее
```

- Как мог бы выглядеть default\_delete?

# Дефолтный удалитель

- Разумеется по дефолту это пустой класс с перегруженными круглыми скобками.

```
template <typename T> struct default_delete {  
    void operator() (T *ptr) { delete ptr; }  
};
```

- Теперь давайте вернемся к исходной проблеме.

```
class MyClass; // предварительное объявление  
  
struct MySafeWrapper {  
    unique_ptr<MyClass> c;  
    MySafeWrapper() : c(nullptr) {}; // увы, не компилируется  
};
```

# Решение: пользовательский делетер

- Внезапно нам помогает пользовательский удалитель.

```
class MyClass; // предварительное объявление

struct MyClassDeleter {
    void operator()(MyClass *); // определён где-то ещё
};

struct MySafeWrapper {
    unique_ptr<MyClass, MyClassDeleter> c;
    MySafeWrapper() : c(nullptr) {} // ok
};
```

- В данном случае проблема была в том, что `delete` проникает в хедер при использовании стандартного удалителя.

# Обсуждение: unique void pointer

- Вспомним про рассмотренный ранее unique pointer.
- Может ли он работать как умный void pointer для стирания типов?
- В чистом виде это невозможно даже скомпилировать.

`unique_ptr<void> u;`

- Можно ли разумно модифицировать это определение?

# Тизер: влияние на размеры

- Как вы думаете, влияет ли на размер необходимость хранить удалитель?

# Обсуждение

- Вернемся к базовой арифметике.
- Итак, мы умеем определять унарные плюс/минус.
- Какие ещё арифметические операторы в языке вы можете вспомнить?

# Источник названия языка

- Язык C++ получил название от операции ++ (постинкремента).
- Бывает также преинкремент.

```
int x = 42, y, z;  
y = ++x; // y = 43, x = 43  
z = y++; // z = 43, y = 44
```

- Для их переопределения используется один и тот же operator++.

```
Quat<T>& Quat<T>::operator++(); // это пре или пост?
```

# Источник названия языка

- Язык C++ получил название от операции ++ (постинкремента).
- Бывает также преинкремент.

```
int x = 42, y, z;  
y = ++x; // y = 43, x = 43  
z = y++; // z = 43, y = 44
```

- Для их переопределения используется один и тот же operator++.

```
Quat<T>& Quat<T>::operator++(); // это pre-increment
```

```
Quat<T> Quat<T>::operator++(int); // это post-increment
```

- Дополнительный аргумент в постинкременте липовый.

# Источник названия языка

- Обычно постинкремент делается в терминах преинкремента.

```
template<typename T> struct Quat {  
    T x_, y_, z_, w_;  
  
    Quat<T>& Quat<T>::operator++() { x_ += 1; return *this; }  
  
    Quat<T> Quat<T>::operator++(int) {  
        Quat<T> tmp {*this};  
        ++(*this);  
        return tmp;  
    }  
};
```

- Разумеется точно так же работает декремент и постдекремент.

# Обсуждение: немного джигитовки

- Признак новичка это "неэффективный" обход контейнера.

```
using itt = typename my_container<int>::iterator;  
for (itt it = cont.begin(); it != cont.end(); it++) {  
    // do something  
}
```

- Профессионал использует преинкремент и не будет делать вызовов в проверке условия.

```
for (itt it = cont.begin(), ite = cont.end(); it != ite; ++it) {  
    // do something  
}
```

# Цепочечные операторы

- Операторы, образующие цепочки имеют вид op=.

```
int a = 3, b = 4, c = 5;
```

```
a += b *= c -= 1; // чему теперь равны a, b, c?
```

- Все они правоассоциативны.
- Исключение составляют очевидные бинарные >= и <=.
- Все они модифицируют свою правую часть и их место внутри класса в качестве его методов.

# Цепочные операторы

- Например для кватернионов:

```
struct Quat {  
    int x, y, z, w;  
  
    Quat& operator+=(const Quat& rhs) {  
        x += rhs.x; y += rhs.y; z += rhs.z; w += rhs.w;  
        return *this;  
    }  
};
```

- Здесь возврат ссылки на себя нужен чтобы организовать цепочку.

```
a += b *= c; // a.operator+=(b.operator*(c));
```

# Определение через цепочки

- Чем плоха идея теперь определить в классе и оператор +?

```
struct Quat {  
    int x, y, z, w;  
  
    Quat& operator+=(const Quat& rhs);  
  
    Quat operator+(const Quat& rhs) {  
        Quat tmp(*this); tmp += rhs; return tmp;  
    }  
}
```

- Казалось бы всё хорошо:

```
Quat x, y; Quat t = x + y; // ok?
```

- ❑ Приведение типов
- ❑ Операторы и цепочки
  - Бинарные операторы
- ❑ Экзотика

# Неявные преобразования

- Часто мы хотим чтобы работали неявные преобразования.

```
Quat::Quat(int x);
```

```
Quat Quat::operator+(const Quat& rhs);
```

```
Quat t = x + 2; // ok, int -> Quat
```

```
Quat t = 2 + x; // FAIL
```

- Увы, метод класса не преобразует свой неявный аргумент.
- Единственный вариант делать настоящие бинарные операторы это делать их вне класса.

# Неявные преобразования

- Часто мы хотим чтобы работали неявные преобразования.

```
Quat::Quat(int x);
```

```
Quat operator+(const Quat& lhs, const Quat& rhs);
```

```
Quat t = x + 2; // ok, int -> Quat rhs
```

```
Quat t = 2 + x; // ok, int -> Quat lhs
```

- Увы, метод класса не преобразует свой неявный аргумент.
- Единственный вариант делать настоящие бинарные операторы это делать их вне класса.

# Определение через цепочки

- Это не мешает использовать для определения бинарных операторов цепочечные с соответствующими аргументами.

```
Quat operator+(const Quat& x, const Quat& y) {  
    Quat tmp {x};  
    tmp += y;  
    return tmp;  
}
```

- Это логично и позволяет переиспользовать код.
- Кроме того такой оператор может не быть friend и действовать в терминах открытого интерфейса.

# Призыв к осторожности

- Одновременное наличие implicit ctors и внешних операторов может вызывать странные эффекты.

```
struct S {  
    S(std::string) {}  
    S(std::wstring) {}  
};  
  
bool operator==(S lhs, S rhs) { return true; }  
  
assert (std::string{"foo"} == std::wstring{L"bar"}); // WAT?
```

- В таких случаях стоит рассмотреть возможность занести сравнение внутрь и сделать его friend.

# Одна небольшая проблема

- Увы это не работает для шаблонов.

```
template<typename T>
Quat<T> operator+ (const Quat<T>& x, const Quat<T>& y) {
    Quat<T> tmp {x};
    tmp += y;
    return tmp;
}
```

- Такой оператор будет скорее всего иметь проблемы с подстановкой типов.
- Потому что преобразование не работает через шаблонную подстановку.

# Сравнение для basic\_string

- Принятый (в т.ч. в libstdc++) вариант решения использует перегрузки.

```
template<typename CharT, typename Traits, typename Alloc>
bool operator==(const basic_string<CharT, Traits, Alloc>& lhs,
                  const basic_string<CharT, Traits, Alloc>& rhs) {
    return lhs.compare(rhs) == 0;
}

template<typename CharT, typename Traits, typename Alloc>
bool operator==(const CharT* lhs, const basic_string<CharT, Traits, Alloc>& rhs) {
    return rhs.compare(lhs) == 0;
}

template<typename CharT, typename Traits, typename Alloc>
bool operator==(const basic_string<CharT, Traits, Alloc>& lhs, const CharT* rhs) {
    return lhs.compare(rhs) == 0;
}
```

# Обсуждение

- *Должен ли* оператор сложения действительно складывать?
- *Должен ли* он быть согласован с цепочечным `+=?`

# Обсуждение

- *Должен ли* оператор сложения действительно складывать?
- *Должен ли* он быть согласован с цепочечным `+=?`
- Увы, на оба вопроса правильный ответ нет.
- Хорошим тоном является поддерживать консистентную семантику, но никто не заставляет делать это.
- В языках с перегрузкой операторов вы никогда не можете быть уверены что делает сложение сегодня утром.
- Поэтому во многих языках этой опции сознательно нет.

# Интермедиа: невезучий сдвиг

- Меньше всего повезло достойному бинарному оператору сдвига.

```
int x = 0x50;  
int y = x << 4; // y = 0x500  
x >>= 4; // x = 0x5
```

- У него, как видите, даже есть цепочечный эквивалент.
- Но сейчас де-факто принято в языке использовать его для ввода и вывода на поток и именно в бинарной форме.

```
std::cout << x << " " << y << std::endl;  
std::cin >> z;
```

# Интермедиа: невезучий сдвиг

- Обычно сдвиг делают всё-таки вне класса используя внутренний дамп.

```
template<typename T> struct Quat {  
    T x, y, z, w;  
    void dump(std::ostream& os) const {  
        os << x << " " << y << " " << z << " " << w;  
    }  
};
```

- И далее собственно оператор (тут не лучшая его версия).

```
template <typename T>  
std::ostream& operator<<(std::ostream& os, const Quat<T>& q) {  
    q.dump(os); return os;  
}
```

# Обсуждение

- А что насчёт сигнатуры?
- Она хотя бы должна быть правильной?

# Обсуждение

- А что насчёт сигнатур?
- Она хотя бы должна быть правильной?
- С точностью до количества аргументов. У бинарного оператора это:
  1. (a).operatorX(b)
  2. operatorX(a, b)
- У оператора присваивания и некоторых других есть только первая форма.
- С точки зрения языка и operator= и operator+ и operator+= это независимые бинарные операторы. По сути просто разные методы.

# Проблемы определения через цепочки

- Для матриц не всё так красиво.

```
template <typename T> class Matrix {  
    // .....,  
    Matrix &operator+=(const Matrix& rhs);  
};  
  
Matrix operator+(const Matrix& lhs, const Matrix& rhs) {  
    Matrix tmp{lhs}; tmp += rhs; return tmp;  
}
```

- Здесь создаётся довольно дорогой временный объект.

```
Matrix x = a + b + c + d; // а здесь трижды
```

# Обсуждение

- Должны ли мы сохранять основные математические свойства операций?
- Например умножение для всех встроенных типов коммутативно.
- Имеет ли смысл тогда переопределять operator\* для матриц?
- Или оставить его только для умножения матрицы на число?

# Сравнения как бинарные операторы

- В чём отличие следующих двух способов сравнить кватернионы?

```
// 1
template<typename T>
bool operator== (const Quat<T>& lhs, const Quat<T>& rhs) {
    return (&lhs == &rhs);
}

// 2
template<typename T>
bool operator== (const Quat<T>& lhs, const Quat<T>& rhs) {
    return (lhs.x == rhs.x) && (lhs.y == rhs.y) &&
           (lhs.z == rhs.z) && (lhs.w == rhs.w);
}
```

# Равенство и эквивалентность

- Базовая эквивалентность объектов означает что их адреса равны (то есть это **один и тот же объект**).
- Равенство через operator== может работать сколь угодно сложно.

```
bool operator== (const Foo& lhs, const Foo& rhs) {  
    bool res;  
    std::cout << lhs << " vs " << rhs << "?" << std::endl;  
    std::cin >> std::boolalpha >> res;  
    return res;  
}
```

- Это конечно крайний случай, но почему нет.

# Равенство и эквивалентность

- Считается, что хороший оператор равенства удовлетворяет трём основным соотношениям.

```
assert(a == a);
```

```
assert((a == b) == (b == a));
```

```
assert((a != b) || ((a == b) && (b == c)) == (a == c));
```

- Первое это рефлексивность, второе симметричность, третье транзитивность.
- Говорят что обладающие такими свойствами отношения являются **отношениями эквивалентности**.

# Двумерные сравнения

- В языке С приняты тривалентные сравнения.

```
strcmp(p, q); // returns -1, 0, 1
```

- В языке C++ приняты двувалентные сравнения.

```
if (p > q) // if (strcmp(p, q) == 1)  
if (p >= q) // if (strcmp(p, q) != -1)
```

- Кажется из одного тривалентного сравнения  $\leq \geq$  можно соорудить все двухвалентные.

# Spaceship operator

- В 2020 году в C++ появился перегружаемый "оператор летающая тарелка".

```
struct MyInt {  
    int x_;  
    MyInt(int x = 0) : x_(x) {}  
    std::strong_ordering operator<=>(const MyInt &rhs) {  
        return x_ <=> rhs.x_;  
    }  
};
```

- Такое определение MyInt сгенерирует все сравнения кроме равенства и неравенства (потому что он не сможет решить какое вы хотите равенство).

# Spaceship operator

- Самое важное это концепция упорядочения.

```
struct S {  
    ordering type operator<=>(const S& that) const
```

- Всего доступны три вида упорядочения.

Тип упорядочения	Равные значения	Несравнимые значения
std::strong_ordering	Неразличимы	Невозможны
std::weak_ordering	Различимы	Невозможны
std::partial_ordering	Различимы	Возможны

# Подумайте о копировании

```
struct SubSet {  
    int a, b;  
  
    ??? operator <=>(const SubSet& rhs) {  
        return a <=> rhs.a;  
    }  
};
```

- Простое правило: подумайте что мы копируем и что мы сравниваем.
- Копируем ровно то что сравниваем: strong, сравниваем подмножество: weak, копируем одно, а сравниваем другое, то тип плохо спроектирован.
- Из этого правила есть исключения (case-insensitive compare для строк). Но оно полезно чтобы начать думать про ordering.

# Defaulted spaceship operator

- Летающая тарелка это один из немногих примеров осмысленного умолчания.

```
struct MyInt {  
    int x_;  
    MyInt(int x = 0) : x_(x) {}  
    auto operator<=>(const MyInt &rhs) const = default;  
};
```

- Сгенерированный по умолчанию (изо всех полей класса) он сам определяет упорядочение и как бонус определяет также равенство и неравенство.
- Логика тут такая: если вы генерируете всё по умолчанию, то вы **точно не хотите от равенства ничего необычного**.

# Интермедиа: std::optional

- Используется для альтернативного хранения.

```
std::optional<std::string> create();
```

...

```
auto x = create();
if (x == "Special") dospecial();
if (x != std::nullopt) use(x.value());
```

- Как бы вы реализовали сравнения для этого класса?

```
template <typename T> struct optional {
    bool has_value() const;
    const T& value() const;
    ...
```

# Что мы хотим сравнивать

- ==, !=, <, <=, >-, > для optional<T> и optional<U>
- ==, !=, <, <=, >-, > для optional<T> и U
- ==, !=, <, <=, >-, > для T и optional<U>
- ==, !=, <, <=, >-, > для optional<T> и std::nullopt
- ==, !=, <, <=, >-, > для std::nullopt и optional<U>
- Это, на секундочку, тридцать перегрузок операторов. Можем ли мы сделать лучше, использовав <=>?

- ❑ Приведение типов
  - ❑ Операторы и цепочки
  - ❑ Бинарные операторы
- Экзотика

# Взятие адреса

- Может быть перегружено так же, как разыменование.

```
template <typename T> class scoped_ptr {  
    T *ptr;  
public:  
    scoped_ptr(T *ptr) : ptr{ptr} {}  
    ~scoped_ptr() { delete ptr; }  
    T** operator&() { return &ptr; }  
    T operator*() { return *ptr; }  
    T* operator->() { return ptr; }
```

- В реальности перегружается редко.

# Обсуждение

- А что если мне и правда нужен именно адрес объекта а у него как назло перегружен оператор взятия адреса?

# Ограничения

- Операторы разыменования (\*) и разыменования с обращением (->) обязаны быть методами, они не могут быть свободными функциями, как и operator=
- Какие последствия могло бы иметь разрешение перегружать их как не-методы?
- Очень интересно, что это не относится к разыменованию с обращением по указателю на метод (->\*)
- Кстати, а что это такое?

# Указатели на методы классов

- Имеет ли смысл выражение "указатель на нестатический метод"?

```
struct MyClass { int DoIt(float a, int b) const; };
```

- Казалось бы нет, но, как мы уже говорили, метод **частично** ведёт себя **как будто** это функция вроде такой:

```
int DoIt(MyClass const *this, float a, int b);
```

- Метод класса **не является** этой функцией.
- Например в точке вызова на него должны распространяться соображения времени жизни и контроля доступа.
- Безопасно ли вызывать через указатель на такую функцию?

# Указатели на методы классов

- Имеет ли смысл выражение "указатель на нестатический метод"?

```
struct MyClass { int DoIt(float a, int b) const; };
```

- На удивление да.

```
using constif_t = int (MyClass::*)(float, int) const;
```

- Поддерживается два синтаксиса вызова.

```
constif_t ptr = &MyClass::DoIt;  
MyClass c; (c.*ptr)(1.0, 1);  
MyClass *pc = &c; (pc->*ptr)(1.0, 1);
```

- И второй из них даже перегружается.

# Волшебные свойства ->\*

- Оператор ->\* примечателен своим никаким приоритетом и никакими требованиями к перегрузке.
- Как следствие его где только не используют (приведённый ниже пример чуточку безумный).

```
template <typename T> T& operator->*(pair<T,T> &l, bool r) {  
    return r ? l.second : l.first;  
}  
  
pair<int, int> y {5, 6};  
y ->* false = 7;
```

# Оператор запятая

- Малоизвестен но встречается оператор запятая.

```
for (int i = 0, j = 0; (i + j) < 10; i++, j++) { use(i, j); }
```

- Оператор имеет общий вид:

```
result = foo(), bar();
```

- Здесь выполняется `foo`, потом `bar`, потом в `result` записывается результат `bar`.

```
buz(1, (2, 3), 4); // вызовет buz(1, 3, 4)
```

- Удивительно, но этот оператор тоже перегружается. Раньше этого не делали из боязни потерять sequencing.

# Интермедиа: sequencing

- Выражения, разделённые точкой с запятой состоят в **отношениях последования** sequenced-after и sequenced-before.

```
foo(); bar(); // foo sequenced before bar
```

- Но увы, вызов функции не определяет sequencing.

```
buz(foo(), bar()); // no sequencing between foo and bar
```

- Почему это так важно? Потому что unsequenced modification это UB case.

```
y = x++ + x++; // operator++ and operator++ unsequenced
```

- В этом примере компилятор имеет право отформатировать жёсткий диск. Он вряд ли это сделает, но ситуация неприятная.

# Что нельзя перегрузить

- Доступ через точку `a.b`
- Доступ к члену класса через точку `a.*b`
- Доступ к пространству имён `a::b`
- Последовательный доступ `a ; b`
- **Почти все специальные операторы в том числе `sizeof`, `alignof`, `typeid`**
  - Правило такое: если вы видите специальный оператор, **скорее всего** его нельзя перегрузить
  - Сюда же относятся: `static_cast` и его друзья
- Тернарный оператор `a ? b : c`

# Что не следует перегружать

- Длинные логические операции `&&` и `||` потому что они теряют сокращённое поведение.

`if (p && p->x) // может взорваться если && перегружено`

- Запятую, чтобы не потерять sequencing в младших стандартах (допустим в примере ниже foo инициализирует данные, которые использует bar).

`x = foo(), bar(); // может взорваться если , перегружена`

- Унарный плюс, чтобы не потерять positive hack.

# И это ещё не всё

- Фундаментальную роль в языке играют операторы работы с памятью и их перегрузка: мы по ряду причин пока ничего не сказали про `operator new`, `operator delete` и прочие прекрасные вещи.
- Также по ряду причин на будущее отложено обсуждение оператора "" нужного для [пользовательских литералов](#).
- Начиная с C++20 можно также перегрузить оператор `co_await`.

# Литература

- [CC11] ISO/IEC 14882 – "Information technology – Programming languages – C++", 2011
- [BS] Bjarne Stroustrup – The C++ Programming Language (4th Edition), 2013
- [GB] Grady Booch – Object-Oriented Analysis and Design with Applications, 2007
- [JG] Joshua Gerrard – The dangers of C-style casts, CppCon, 2015
- [BD] Ben Deane – Operator Overloading: History, Principles and Practice, CppCon, 2018
- [JM] Jonathan Müller – Using C++20's Three-way Comparison <=>, CppCon, 2019
- [TW] Titus Winters – Modern C++ Design, CppCon, 2018