ЛЕКЦИЯ 16
ПЕРЕГРУЗКА ОПЕРАТОРОВ

АЛГОРИТМИЗАЦИЯ И ПРОГРАММИРОВАНИЕ

ЛЕКТОР ФУРМАВНИН С.А.

ВАШИ ТИПЫ КАК ВСТРОЕННЫЕ

Собственный класс трехмерного вектора

```
struct Vec3D {
    int x, y, z;
};
```

У нас уже есть бесплатное копирование и присваивание. Хотелось бы чтобы работало всё остальное: сложение, умножение на число и так далее

Начнем с чего-то простого

```
Vec3D q{1, 2, 3};
Vec3D p = -q; // унарный минус : {-1, -2, -3}
```

ОБЩИЙ СИНТАКСИС ОПЕРАТОРОВ

Обычно используется запись operator и далее какой это оператор

```
struct Vec3D {
    int x, y, z;
};
Vec3D operator-(Vec3D arg) {
    return Vec3D{ -arg.x, -arg.y, -arg.z};
}
```

Теперь всё как надо

```
Vec3D q{1, 2, 3};
Vec3D p = -q; // унарный минус : {-1, -2, -3}
```

ОБЩИЙ СИНТАКСИС ОПЕРАТОРОВ

Альтернатива – метод класса

```
struct Vec3D {
    int x, y, z;
    Vec3D operator-() {
        return Vec3D{ -x, -y, -z};
}
};
```

И снова всё как надо

```
Vec3D q{1, 2, 3};
Vec3D p = -q; // унарный минус : {-1, -2, -3}
```

ОБСУЖДЕНИЕ

Обычно есть пара вариантов (исключение: присваивание и пара-тройка других)

- `-a` означает `a.operator-()`
- `-a` 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;
}
```

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; }

// не эффективно: вызовы по указателям
std::sort(myarr.begin(), myarr.end(), &gtf);
```

Можно ли с этим что-то сделать?

ФУНКТОРЫ: ПЕРВЫЙ ВАРИАНТ РЕШЕНИЯ

Функтором называется класс, который ведет 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 выглядит жутковато и плохо расширяется

ФУНКТОРЫ: ПЕРЕРЕГРУЗКА()

Более правильный способ сделать функтор – это перегрузка вызова

```
struct gt {
    bool operator()(int x, int y) { return x > y; }
};

// все также 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 дойдем 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.

Хорошей ли идеей является заменить здесь `CImpl*` на `std::unique_ptr`?

КОРОТКО ОБ ABI

ABI — это набор соглашений, которые описывают двоичный интерфейс приложения. В частности, он регламентирует использование стека и регистров процессора, порядок передачи аргументов и возвращаемого значения для функций, размеры базовых типов и многое другое. В целом, ABI описывает детали реализации взаимодействия приложений между собой и между операционной системой (платформой).

ПРОБЛЕМА НЕПОЛНОГО ТИПА

Попробуем использовать unique pointers в PImpl

```
class MyClass; // forward declaration

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

struct MySafeWrapper {
    std::unique_ptr<MyClass> c; // увы, это CE
    MySafeWrapper() : c(nullptr) {};
};
```

КАК РЕАЛЬНО ВЫГЛЯДИТ UNIQUE_PTR?

Стратегия удаления вынесена у него 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 может выглядеть `default_delete`?

DEFAULT_DELETE

Разумеется по умолчанию это пустой класс с перегруженным оператором круглые скобки

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

Теперь вернемся к исходной проблеме

```
class MyClass; // forward declaration

struct MySafeWrapper {
    std::unique_ptr<MyClass> c; // увы, это CE
    MySafeWrapper() : c(nullptr) {};
};
```

РЕШЕНИЕ: ПОЛЬЗОВАТЕЛЬСКИЙ ДЕЛЕТЕР

Внезапно нам помогает пользовательский удалитель

```
class MyClass; // forward declaration

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

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

UNIQUE VOID POINTER

Вспомним про рассмотренный ранее unique_ptr

Может ли он работать, как умный void pointer для стирания типов?
В чистом виде это невозможно даже скомпилировать

```
std::unique_ptr<void> u;
```

Можно ли разумно модифицировать это определение?

ВСПОМНИМ ПРО РАЗМЕР

Как вы думаете, влияет ли на размер необходимость хранить делетер?

BACK TO BASICS

Вернемся к базовой арифметике

Итак, мы умеем определять унарные плюс/минус

Какие еще арифметические операторы в языке вы можете вспомнить?

ИСТОЧНИК НАЗВАНИЯ ЯЗЫКА

Язык C++ получил называние от операции ++(постинкремента)
Бывает также преинкремент

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

Для их определения используется один и тот же operator++

```
Vec3D& Vec3D::operator++(); // это пре- или пост-?
```

ИСТОЧНИК НАЗВАНИЯ ЯЗЫКА

Язык C++ получил называние от операции ++(постинкремента)
Бывает также преинкремент

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

Для их определения используется один и тот же operator++

```
Vec3D& Vec3D::operator++(); // preincrement
Vec3D Vec3D::operator++(int); // postincrement
```

Дополнительный аргумент в постинкременте липовый (dummy)

ИСТОЧНИК НАЗВАНИЯ ЯЗЫКА

Обычно постинкремент делается в терминах преинкремента

```
struct Vec3D {
    int x, y, z;
    Vec3D& operator++() { x += 1; return *this; }
    Vec3D& operator++(int) {
        Vec3D tmp {*this};
        ++(*this);
        return tmp;
    }
};
```

Разумеется, точно также работают предекремент и постдекремент

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 и c?
```

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

ЦЕПОЧЕЧНЫЕ ОПЕРАТОРЫ

Например, для нашего вектора

```
struct Vec3D {
    int x, y, z;
    Vec3D& operator+=(const Vec3D& rhs) {
        x += rhs.x; y += rhs.y; z += rhs.z;
        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 `operator+`?

```
struct Vec3D {
    int x, y, z;
    Vec3D& operator+=(const Vec3D& rhs);
    Vec3D operator+(const Vec3D& rhs) {
        Vec3D tmp(*this); tmp += rhs; return tmp;
    }
};
```

Казалось бы, всё хорошо:

```
Vec3D x, y; Vec3D z = x + y; // ok?
```

НЕЯВНЫЕ ПРЕОБРАЗОВАНИЯ

Часто мы хотим, чтобы работали и неявные преобразования

```
Vec3D::Vec3D(int x);
Vec3D Vec3D::operator+(const Vec3D& rhs);

Vec3D t = x + 2; // ok, int -> Vec3D

Vec3D p = 2 + x; // FAIL
```

Увы, метод в классе не преобразует свой неявный аргумент

Единственный вариант делать настоящие бинарные операторы – это делать их вне класса

НЕЯВНЫЕ ПРЕОБРАЗОВАНИЯ

Часто мы хотим, чтобы работали и неявные преобразования

```
Vec3D::Vec3D(int x);
Vec3D operator+(const Vec3D& lhs, const Vec3D& rhs);

Vec3D t = x + 2; // ok, int -> Vec3D

Vec3D p = 2 + x; // ok, int -> Vec3D
```

Увы, метод в классе не преобразует свой неявный аргумент

Единственный вариант делать настоящие бинарные операторы – это делать их вне класса

СПОЙЛЕР ПРО ШАБЛОНЫ

Увы, с шаблонами классов это не работает

БУДЬТЕ ОСТОРОЖНЫ

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”)); // WTF?
```

В таких случаях стоит рассмотреть возможность завести сравнение внутрь и сделать его `friend`

ОБСУЖДЕНИЕ

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

ОБСУЖДЕНИЕ

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

- Увы, на оба вопроса правильный ответ: **НЕТ**
- Хорошим тоном является поддерживать консистентную семантику, но никто не заставляет вас это делать
- В языках с перегрузкой операторов вы никогда не можете быть уверены, что делает сложение сегодня утром
- Поэтому во многих языках программирования этой опции сознательно нет

НЕВЕЗУЧИЙ СДВИГ

- Меньше всего повезло достойному оператору битового сдвига

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

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

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

НЕВЕЗУЧИЙ СДВИГ

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

```
struct Vec3D {
    int x, y, z;
    void dump(std::ostream& os) const {
        os << "(" << x << "," << y << "," << z << ")";
    }
}
```

И далее сам оператор (не лучшая его ипостасия)

```
std::ostream& operator<<(std::ostream& os, const Vec3D v) {
    v.dump(os); return os;
}
```

ОБСУЖДЕНИЕ

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

ОБСУЖДЕНИЕ

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

ПРОБЛЕМЫ ОПРЕДЕЛЕНИЯ ЧЕРЕЗ ЦЕПОЧКИ

Для матриц всё не так красиво

```
class Matrix {
    // ….
    Matrix& operator+=(const Matrix& rhs);
};

Matrix operator+(const Matrix& lhs, const Matrix& rhs) {
    Matrix tmp(lhs); tmp += rhs; return tmp;
}
```

Здесь создается довольно дорогой вариант

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

ОБСУЖДЕНИЕ

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

СРАВНЕНИЯ КАК БИНАРНЫЕ ОПРЕАТОРЫ

- В чем отличие следующих двух способов сравнить векторы?

```
// 1
bool operator==(const Vec3D& lhs, const Vec3D& rhs) {
    return (&lhs == &rhs);
}

// 2
bool operator==(const Vec3D& lhs, const Vec3D rhs) {
    return (lhs.x == rhs.x) && (lhs.y == rhs.y) && (lhs.z ==
    lhz.z);
}
```

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::boolaplha >> res;
    return res;
}
```

Это, конечно, через чур, но почему бы и 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**

ДВУ И ТРИВАЛЕНТНЫЕ СРАВНЕНИЯ

- В языке C 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, что из одного тривалентного сравнения <=> можно соорудить все двувалентные

ОПЕРАТОР КОСМИЧЕСКИЙ КОРАБЛЬ

- В 2020 году в C++ появился перегружаемый «оператор космический корабль»

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

Такое определение `MyInt` сгенерирует все сравнения, кроме равенства и неравенства (потому что он не сможет решить что вы хотите: равенство или эквивалентность)

ОПЕРАТОР КОСМИЧЕСКИЙ КОРАБЛЬ

- Самое важное – это концепция упорядочивания

```
struct S {
    operating type operator <=>(const S& that) const
};
```

Всего доступны три вида упорядочивания

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

КОСМИЧЕСКИЙ КОРАБЛЬ ПО УМОЛЧАНИЮ

- Космический корабль – 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 упорядочивание 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 странного.**

ВЗЯТИЕ АДРЕСА

Может быть перегружено также, как разыменовывание

```
class scoped_ptr {
    S* ptr;
public:
    scoped_ptr(S *ptr) : ptr(ptr) {}
    ~scoped_ptr() {delete ptr;}
    S** operator&() {return &ptr;}
    S operator*() {return *ptr;}
    S* 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 разыменовывания (*) и разыменовывания с обращением (->) обязаны быть методами, они не могут быть свободными функциями, как и `operator=`.

- Какие последствия могло бы иметь разрешение перегружать их как неметоды?

- Очень интересно, что это не относится к разыменовыванию с обращением по указателю на метод (->*)

- Кстати, а что это вообще 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
- Как мы уже говорили, метод **частично** ведёт себя, **как будто** это функция вроде

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

- И на такую функцию возможен указатель. Но метод класса **не является** этой функцией.
- Например, в точке вызова на него должны распространяться соображения времени жизни и контроля доступа. **Вызов через подобный указатель на функцию кажется возможностью нарушить инкапсуляцию.**

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;
```

- Поддерживаются 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**!

ЧУДЕСНЫЕ СВОЙСТВА ->*

- Оператор ->* примечателен своим никаким приоритетом и никакими требованиями к перегрузке
- Как следствие, его где только не используют (приведенный пример слегка безумный)

```
int& operator->*(std::pair<int, int> &l, bool r) {
    return r ? l.second : l.first;
}

std::pair<int, int> y {1, 2};
y ->* false = 7;
```

ОПЕРАТОР ЗАПЯТАЯ

- Малоизвестен, но встречается оператор запятая

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

- Например, он работает в приведенном цикле

- Оператор имеет общий вид

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

- Здесь выполняется сначала `foo()`, потом `bar()`, потом в `result` записывается результат `bar()`

```
buz(1, (3, 4), 5); // вызовет buz(1, 4, 5)
```

- Удивительно, но этот оператор тоже перегружается. **Этого никогда не следует делать**, потому что вы потеряете sequencing!

SEQUENCING

Выражения, разделенные 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 ваш жесткий диск. Он вряд ли это сделает, но ситуация не самая приятная.

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: 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 ленивое поведение

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

- Запятую, чтобы не потерять sequencing (допустим, в примере ниже `foo()` инициализирует данные, которые использует `bar()`;

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

- Унарный плюс, чтобы не потерять positive hack

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 другие 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`

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Bjarne Stroustrup – The C++ Programming Language (4th Edition), 2013
2. Grady Booch – Object-oriented Analysis and Design with Applications, 2007
3. Скотт Мейерс, Эффективный современный C++: 42 способа улучшить ваше использование C++11 и C++14
4. Joshua Gerrard – The dangers of C-style casts, CppCon, 2015
5. Ben Deane – Operator Overloading: History, Principles and Practice, CppCon, 2018
6. Titus Winters – Modern C++ Design, CppCon 2018
7. Дональд Кнут, Искусство программирования. Том 1. Основные алгоритмы / Ю.В. Козаченко. - 3-е изд – Москва, Санкт-Петербург: ВИЛЬЯМС, 2018. – 721 с.
8. Дональд Кнут, Искусство программирования. Том 2. Получисленные алгоритмы / Ю.В. Козаченко. - 3-е изд – Москва, Санкт-Петербург: ВИЛЬЯМС, 2018. – 743 с.
9. Дональд Кнут, Искусство программирования. Том 3. Сортировка и поиск / Ю.В. Козаченко. - 3-е изд – Москва, Санкт-Петербург: ВИЛЬЯМС, 2018. – 767 с.