ОПЕРАТОРЫ

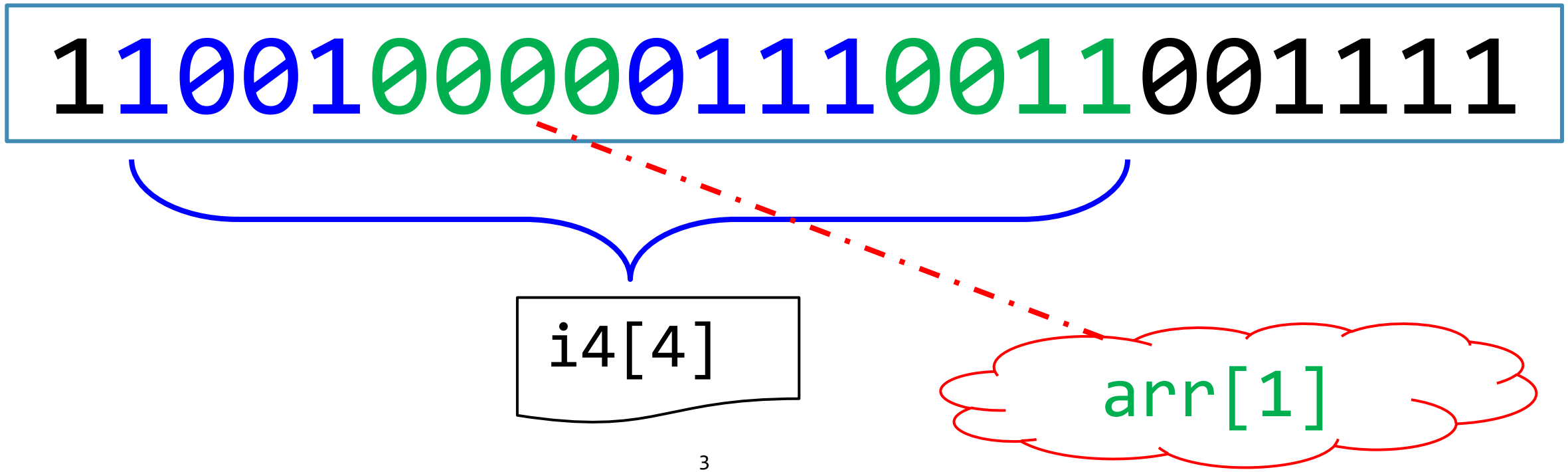
Грязные детали о многомерных массивах. Приведение и перегрузка приведения. Перегрузка основных операторов. Матрицы

К. Владимиров, Intel, 2020
mail-to: konstantin.vladimirov@gmail.com

- Многомерные массивы
- Приведение типов
- Перегрузка операторов
- Проектирование матрицы

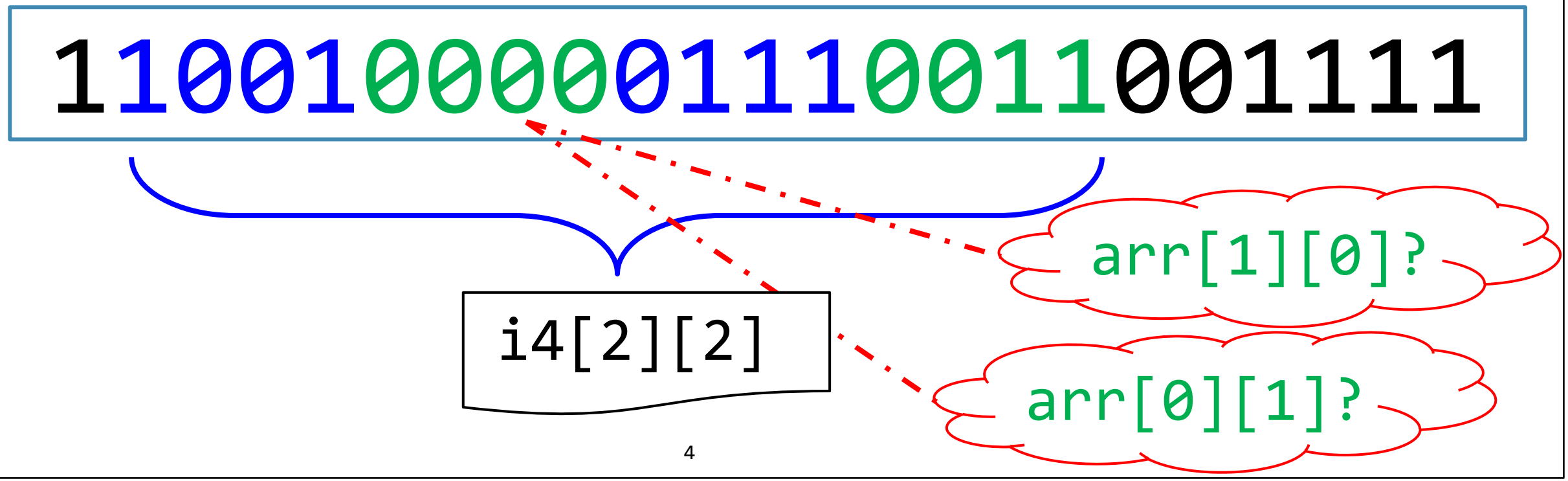
Двумерные массивы

- RAM-модель памяти в принципе одномерна, поэтому с двумерными массивами начинаются сложности



Двумерные массивы

- RAM-модель памяти в принципе одномерна, поэтому с двумерными массивами начинаются сложности

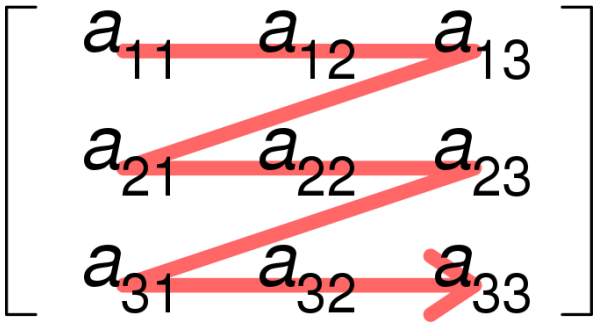


Row-major vs column-major

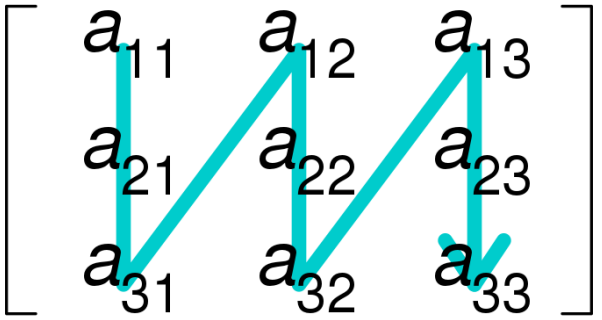
- В математике для матрицы $\{a_{ij}\}$, первый индекс называется индексом строки, второй – индексом столбца
- В языке C принят row-major order (очень просто запомнить: язык C читает матрицы как книжки)
- row-major означает, что первым изменяется самый внешний индекс

```
int one[7]; // 7 столбцов
int two[1][7]; // 1 строка, 7 столбцов
int three[1][1][7]; // 1 слой, 1 строка ..
```

Row-major order



Column-major order



Обсуждение

- Кстати, а кто-нибудь понимает **почему** row-major?

```
int a[7][9]; // declaration follows usage
int elt = a[2][3]; // why 3-rd element of 2-nd row?
```

Обсуждение

- Кстати, а кто-нибудь понимает **почему** row-major?

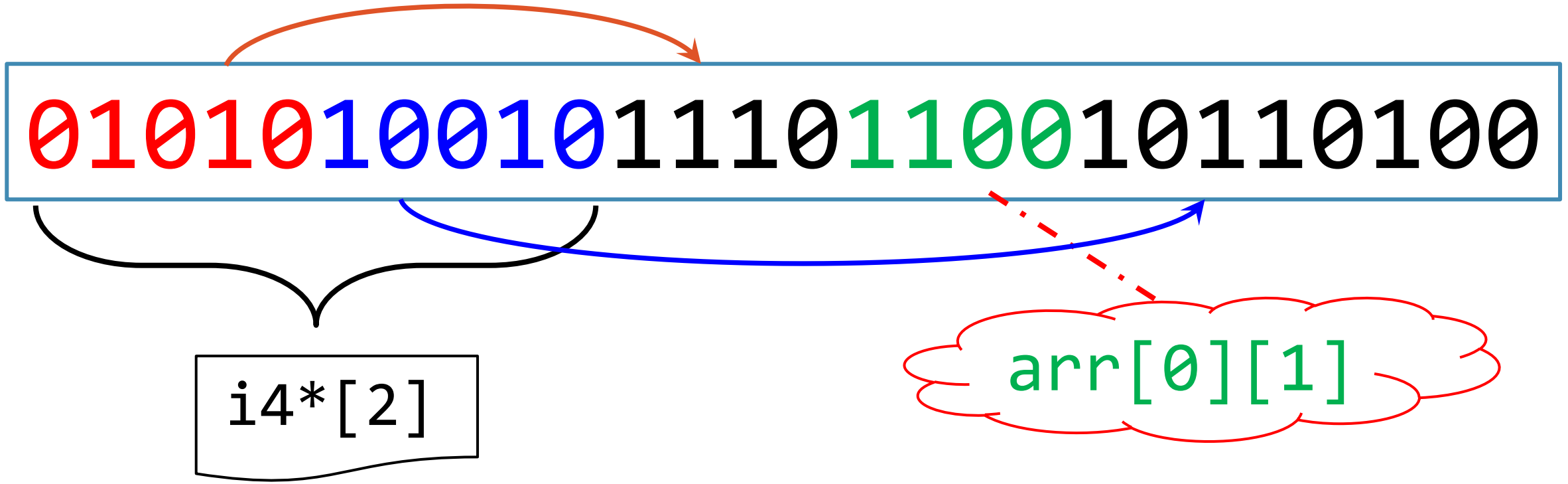
```
int a[7][9]; // declaration follows usage

int elt = a[2][3]; // why 3-rd element of 2-nd row?
```

- Удивительно, но на это есть **синтаксические** причины
- Всё дело в том, что `a[i][j]` это неоднозначное выражение, которое может быть прочитано по разному, в том числе и как `(a[i])[j]`
- Это в свою очередь следует из ещё одного способа представления массивов: представления их как jagged arrays

Двумерные массивы: jagged arrays

- Ещё один способ сделать двумерный массив это сделать массив указателей



Двумерные массивы

- Непрерывный массив

```
int cont[10][10];
foo(cont);
cont[1][2] = 1; // ?
```

- Массив указателей

```
int *jagged[10];
bar(jagged);
jagged[1][2] = 1; // ?
```

- Функция, берущая указатель на массив

```
void foo(int (*pcont)[10]){
  pcont[1][2] = 1; // ?
}
```

- Функция, берущая указатель на массив указателей

```
void bar(int **pjag) {
  pjag[1][2] = 1; // ?
}
```

- Самый интересный вопрос: как во всех четырёх случаях вычисляется доступ к соответствующему элементу?

Вычисление адресов

- Массиво-подобное вычисление

```
int first[FX][FY];
first[x][y] = 3; // → *(&first[0][0] + x * FY + y) = 3;

int (*second)[SY];
second[x][y] = 3; // → *(&second[0][0] + x * SY + y) = 3;
```

- Указателе-подобное вычисление

```
int *third[SX];
third[x][y] = 3; // → *(*(third + x) + y) = 3;

int **fourth;
fourth[x][y] = 3; // → *(*(fourth + x) + y) = 3;
```

Обсуждение

- Сколько индексов можно опускать при инициализации массивов?

```
float flt[2][3] = {{1.0, 2.0, 3.0}, {4.0, 5.0}}; // ok
float flt[][3] = {{1.0, 2.0, 3.0}, {4.0, 5.0}}; // ?
float flt[][] = {{1.0, 2.0, 3.0}, {4.0, 5.0}}; // ?
```

Обсуждение

- Сколько индексов можно опускать при инициализации массивов?

```
float flt[2][3] = {{1.0, 2.0, 3.0}, {4.0, 5.0}}; // ok
float flt[][3] = {{1.0, 2.0, 3.0}, {4.0, 5.0}}; // ok
float flt[][] = {{1.0, 2.0, 3.0}, {4.0, 5.0}}; // fail
```

- Мы всегда можем опускать только самый вложенный индекс: и в инициализаторах и в аргументах функций
- Очень просто запомнить: массивы гниют изнутри

```
float func(float flt[][3][6]); // ok, float *flt[3][6]
```

Corner-case

- Обычно `a[]` означает `*a`, это верно почти всегда
- Увы, есть один случай, когда это не так: объявления

```
extern int *a; // где-то есть настоящая ячейка a
extern int b[]; // где-то есть массив b какой-то длины
```

- Все ли осознают с чем это связано?

Corner-case

- Обычно a[] означает *a, это верно почти всегда
- Увы, есть один случай, когда это не так: объявления

```
extern int *a; // где-то есть настоящая ячейка a
extern int b[]; // где-то есть массив b какой-то длины
```

- Все ли осознают с чем это связано?
- Разумеется не с правилами вычисления

```
i = a[5]; // i = *(a + 5);

i = b[5]; // i = *(b + 5);
```

Corner-case

- Обычно a[] означает *a, это верно почти всегда
- Увы, есть один случай, когда это не так: объявления

```
extern int *a; // где-то есть настоящая ячейка a
extern int b[]; // где-то есть массив b какой-то длины
```

- Все ли осознают с чем это связано?
- Это связано с разной **операционной семантикой**

```
i = a[5]; // aval = load [a];
             i = load [aval + 5 * sizeof(int)]
i = b[5]; // i = load [b + 5 * sizeof(int)]
```

Case study: представление матрицы

- jagged vector

```
struct matrix {
  int **data;
  int x, y;
};
```

- непрерывный массив

```
struct matrix {
  int *data;
  int x, y;
};
```

- Какие вы видите плюсы и минусы в обоих методах?
- Подумайте об умножении матриц и оптимизациях кэш-эффектов
- Подумайте о других операциях, например обмене строк

- Многомерные массивы
  - Приведение типов
- Перегрузка операторов
- Проектирование матрицы

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 всё при этом язык C++ наследует старую добрую линейную модель памяти, в которой любой объект это просто кусок памяти

```
float f = 1.0;

int x = *(int *)&f; // что в x?
```

Обсуждение

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

Обсуждение

- Не имеет ли приведение в стиле C (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
- Но в языке C всё это пишется как

```
x = (T) y;
```

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

```
uintptr_t uq = reinterpret_cast<uintptr_t>(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

04-opers/reinterpret.cc

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
- Увы, есть вещи, которые C++ всё таки унаследовал

Особенности неявного приведения

- В наследство от языка C 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 = 0xFFFE, y = 0xEEEE;
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

- Сишные 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
```

Пользовательские преобразования

- Конструкторы определяют неявное преобразование типа

```
struct MyString {
  char *buf_; size_t len_;
  MyString(size_t len) : buf_{new char[len]{}}, len_{len} {}
};

void foo(MyString);

foo(42); // ok, MyString implicitly constructed
```

- Иногда это очень полезно (например конструкция Quat из int)
- Но это не всегда хорошо, например в ситуации со строкой, мы ничего такого не имели в виду

04-opers/implicit.cc

Требуем ясности

- Ключевое слово `explicit` указывается когда мы хотим заблокировать пользовательское преобразование

```
struct MyString {
  char *buf_; size_t len_;
  explicit MyString(size_t len) :
    buf_{new char[len]{}}, len_{len} {}
};
```

- Теперь здесь будет ошибка компиляции

```
void foo(MyString);

foo(42); // error: could not convert '42' from 'int' to 'MyString'
```

Пользовательские преобразования

- В некоторых случаях мы не можем сделать конструктор. Скажем что если мы хотим неявно преобразовывать `Quat<int>` в `int`?
- Тогда мы пишем operator type

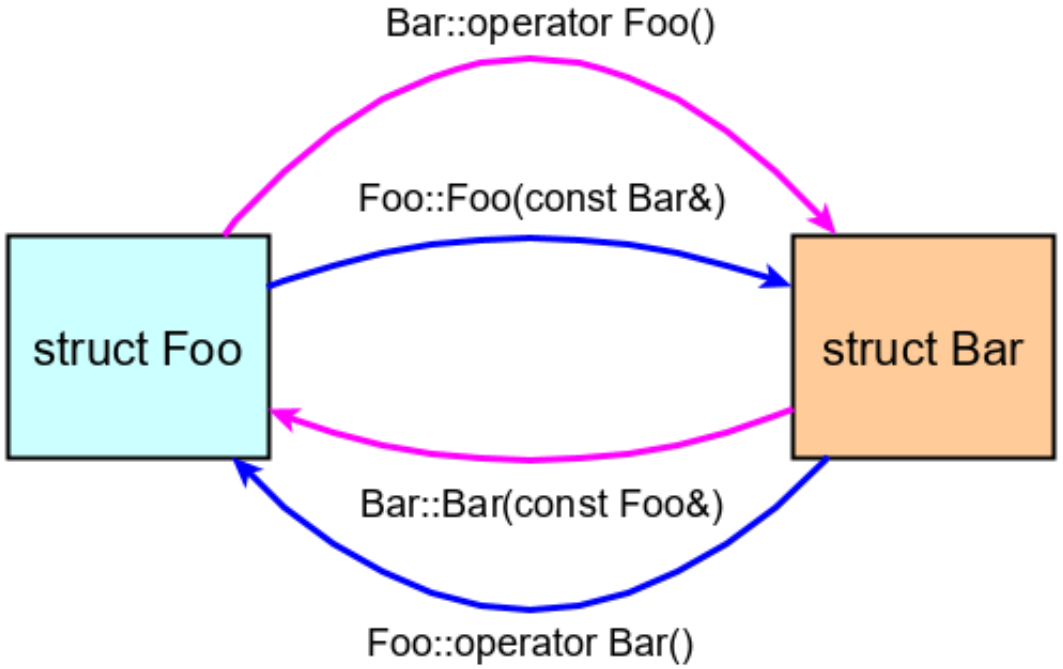
```
template<typename T> struct Quat {
  T x_ = 0, y_ = 0, z_ = 0, w_ = 0;

  operator T() { return x_; } // Quat<T> -> T
```

- Можно `operator int`, `operator double`, `operator S` и так далее
- На такие операторы можно навешивать `explicit` тогда возможно только явное преобразование

Пользовательские преобразования

- Таким образом есть некая избыточность: два способа перегнать туда и два способа перегнать обратно
- Конечно хороший тон это использовать конструкторы где возможно
- Как вы думаете что будет при конфликте?



04-opers/conflict.cc

Перегрузка

- Теперь мы знаем что такое загадочные пользовательские преобразования. Они участвуют в перегрузке и проигрывают стандартным и точному совпадению (но выигрывают у троеточий)

```
struct Foo { Foo(long x = 0) {} };

void foo(int x);

void foo(Foo x);

long l;
foo(l); // вызовет foo(int)
```

Унарный плюс (positive hack)

- Этот оператор интересен тем, что для почти всех встроенных типов он не значит ничего
- 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)
```


04-opers/convert.cc

Обсуждение

- До сих пор мы видели только operator= как вдруг…
- Как вы думаете, а что вообще можно перегружать?

- Многомерные массивы
- Приведение типов
- Перегрузка операторов
- Проектирование матрицы

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)
- `-a` означает `a.operator-()`
- `-a` означает `operator-(a)`
- Как вы думаете, что будет если определить оба?

`04-opers/twovariants.cc`

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;
}
```


04-opers/builtin.cc

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

04-opers/chained.cc

Цепочечные операторы

- Например для кватернионов

```
template<typename T> struct Quat {
  T 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 +?

```
template<typename T> struct Quat {
  T 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<int> x, y; Quat<int> t = x + y; // ok
```

Неявные преобразования

- Часто мы хотим чтобы работали неявные преобразования

```
Quat<T>::Quat<T>(T x);
Quat<T> Quat<T>::operator+(const Quat<T>& rhs);

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

Quat<int> 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<T>::Quat<T>(T x);
Quat<T> operator+(const Quat<T>& lhs, const Quat<T>& rhs);

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

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

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

- Это логично и позволяет переиспользовать код
- Но вот только есть два вопроса….

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
- В языках с перегрузкой операторов вы никогда не можете быть уверены что делает сложение сегодня утром
- Поэтому во многих языках этой опции сознательно нет

Интермедия: 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;
```

Интермедия: 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 сигнатуры?
- Она хотя бы должна быть правильной?
- С точностью до количества аргументов. У бинарного оператора это
- `(a).operatorX (b)`
- `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; // a 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

Дву и три валентные 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 из одного тривалентного сравнения <=> можно соорудить все двухвалентные

Spaceship operator

- В 2020 году в C++ появился "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)

https://godbolt.org/z/7hsE3W

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

Defaulted spaceship operator

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

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

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
}
```

Псевдоуказатели

- Для работы с указателе-подобными объектами можно перегрузить взятие адреса (&) разыменование (*) и обращение с разыменованием (->)

```
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; }
  // как вы думаете а что может возвращать ->?
  // .... тут всё остальное ....
};
```

Псевдоуказатели

- Для работы с указателе-подобными объектами можно перегрузить взятие адреса (&) разыменование (*) и обращение с разыменованием (->)

```
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; } // НО КАК?
  // .... тут всё остальное ....
};
```


04-opers/scoped.cc

Глубже в кроличью нору

- Удивительное поведение стрелочки называется drill-down behavior

```
struct X { int a, b;};

int main() {
  scoped_ptr<X> pt{new X{2, 3}};
  return pt->a; // pt.operator->()->a
}
```

- В реальности вызов `a->b` эквивалентен вызову `(a.operator->())->b`
- Стрелочка как бы "зарывается" в глубину на столько уровней на сколько может

04-opers/drilldown.cc

Обсуждение

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

04-opers/addressof.cc

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
- 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;
```

- 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;
```

Индексаторы

- Допустим мы пишем свой класс похожий на массив

```
class MyVector {
  std::vector<int> v_;
public:
  int& operator[](int x) { return v[x]; }
  const int& operator[](int x) const { return v[x]; }
  // .... some stuff ....
};
```

- Мы хотим его индексировать и для этого перегружаем квадратные скобки
- Перегрузка для const как обычно важна: она даёт возможность работать с const объектом

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?

04-opers/qsortbench.cc

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(x, y); }
};

// гораздо лучше: теперь возможна подстановка
std::sort(myarr.begin(), myarr.end(), gt{});
```

- Увы, это жутковато выглядит и плохо расширяется

04-opers/qsortbench-cast.cc

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

04-opers/qsortbench-op.cc

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 приведённом цикле
- 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. Это никогда не следует делать, потому что вы потеряете sequencing

Интермедия: 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` (здесь явно не место для их полного перечня, но правило такое: если вы видите специальный оператор, скорее всего его нельзя перегрузить)
- Приведения: `static_cast` и его друзей
- 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 (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`

- Многомерные массивы
- Приведение типов
- Перегрузка операторов
- Проектирование матрицы

Проектирование следует применению

- Сначала полезно понять как мы хотим использовать матрицы

```
Matrix m1{5, 6}; // 5 x 6 matrix of all zeros

std::vector<int> v = {0, 1, 1, 0};
Matrix m2{2, 2, v.begin(), v.end()}; // {0, 1; 1, 0}

Matrix m3 = Matrix::eye(2, 2);

Matrix m4 = m3;

m3 = m1; m2 += m1; m4 = m3.transpose() + m2 * 2;

m2.prod_eq(m2); Matrix m5 = prod(m3, m1);
```

Конструкторы

```
template <typename T> class Matrix {
  // некое представление
public:

  // конструктор для создания матрицы, заполненной значением
  Matrix(int cols, int rows, T val = T{});

  // конструктор для создания из заданной последовательности
  template <typename It>
  Matrix(int cols, int rows, It start, It fin);

  // "конструктор" для создания единичной матрицы
  static Matrix eye(int n, int m);
```

Большая тройка

```
template <typename T> class Matrix {
  // некое представление
public:

  // копирующий конструктор
  Matrix(const Matrix &rhs);

  // присваивание
  Matrix& operator=(const Matrix &rhs);

  // деструктор
  ~Matrix();
```

Селекторы

```
template <typename T> class Matrix {
  // некое представление
public:

 // базовые
 int ncols() const;
 int nrows() const;
 const T& operator[]() const;

 // агрегатные
 T trace() const;
 bool equal(const Matrix& other) const;
 void dump(std::ostream& os) const;
```

Удобные методы

```
template <typename T> class Matrix {
  // некое представление
public:

  // отрицание
  Matrix& negate();
  Matrix operator-() const
    { Matrix tmp{this}; return tmp.negate(); }

  // почему не Matrix transpose() const?
  Matrix& transpose();

  // инверсия
  Matrix& invert();
```

Небольшая проблема

- В принципе вот так всё ок:

```
Matrix m{2, 3};
Matrix &mref = m.transpose().negate();
```

- А вот так уже чревато

```
Matrix &mref = (-m).negate();
```

- Что может здесь пойти не так?

Решение: lvalue qualifiers

```
template <typename T> class Matrix {
  // некое представление
public:

  // отрицание
  Matrix& negate() &;
  Matrix operator-() const;

  // почему не Matrix transpose() const?
  Matrix& transpose() &;

  // равенство
  bool equal(const Matrix& other) const;
```


https://godbolt.org/z/8WdPdj

Перегруженные операторы

```
template <typename T> class Matrix {
  // некое представление
public:

  // арифметика цепочками
  Matrix& operator+= (const Matrix& rhs) &;
  Matrix& operator-= (const Matrix& rhs) &;
  Matrix& operator*= (int n) &;
  Matrix& prod_eq(const Matrix& rhs) &;

  // сеттеры
  T& operator[]();
```

Внешние операторы

- При правильном проектировании они обычно тривиальны

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

Matrix operator= (const Matrix& lhs, const Matrix& rhs) {
  return lhs.equal(rhs);
}
```

- И так далее
- Заметьте: ни один из них не испытывает необходимости быть другом

Обсуждение

- Давайте подумаем над вычислением определителя
- Есть тривиальный комбинаторный алгоритм который очень плох
- Есть более сложные например LU-decomposition
- Их преимущества очевидны (скорость), но есть ли у них недостатки?

Домашняя работа HWMX

- Вам предлагается найти определитель матрицы
- На стандартный ввод приходит размер n и далее все элементы построчно
- На стандартном выводе должно быть значение определителя
- Пример
- Вход: `2 1 0 0 1`
- Выход: `1`

Литература

- $[CC11]$ ISO/IEC 14882 – "Information technology – Programming languages – C++", 2011
- $[BS]$ Bjarne Stroustrup – The C++ Programming Language (4th Edition) , 2013
- $[BD]$ Ben Deane – Operator Overloading: History, Principles and Practice, CppCon, 2018
- $[MA]$ Andrew Stuart, Jochen Voss – Matrix Analysis and Algorithms, 2009
- $[GS]$ Gilbert Strang – Introduction to Linear Algebra, Fifth Edition, 2016
- $[JM]$ Jonathan Müller – Using C++20's Three-way Comparison <=>, CppCon, 2019