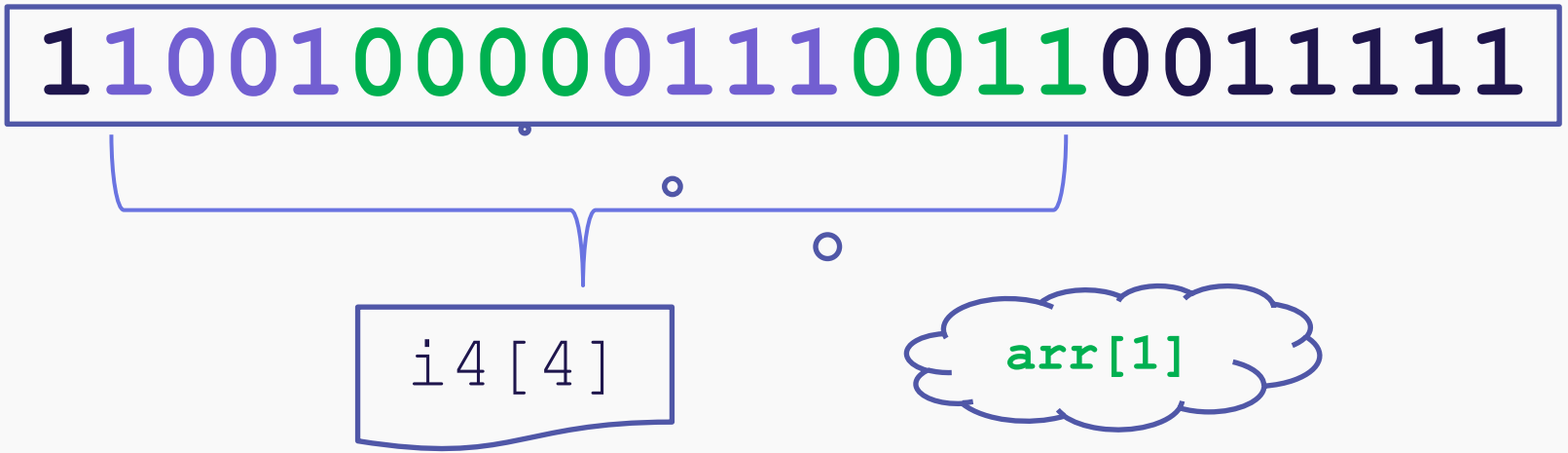
ЛЕКЦИЯ 15
ПРИВЕДЕНИЕ ТИПОВ

АЛГОРИТМИЗАЦИЯ И
ПРОГРАММИРОВАНИЕ

ЛЕКТОР ФУРМАВНИН С.А.

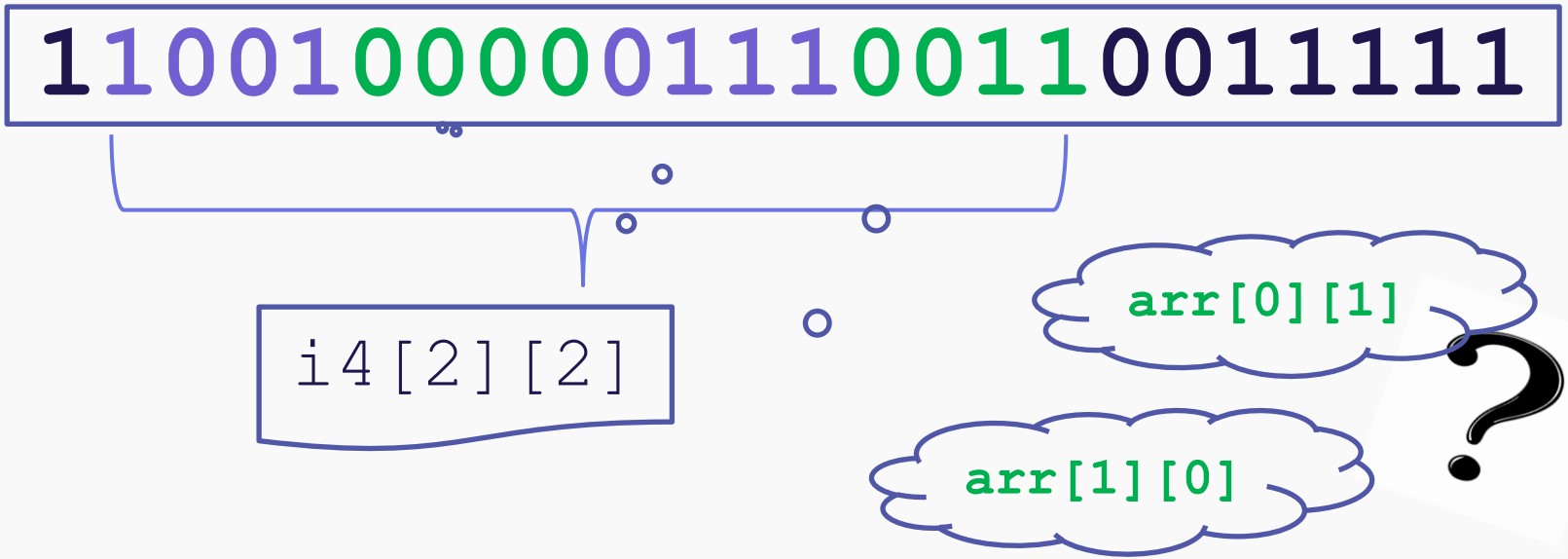
ДВУМЕРНЫЕ МАССИВЫ

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

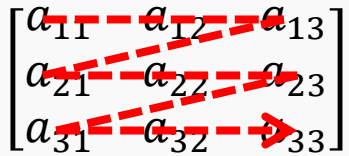


ROW-MAJOR vs COLUMN-MAJOR

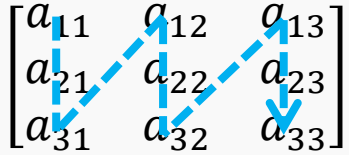
- В математике для матрицы $\{a_{ij}\}$, первый индекс называется индексом строки, второй – индексом столбца
- В языке C принят row-major order (очень просто запомнить: язык C читает матрицы как книжки)
- row-major означает, что первым изменяется самый внешний индекс

```
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



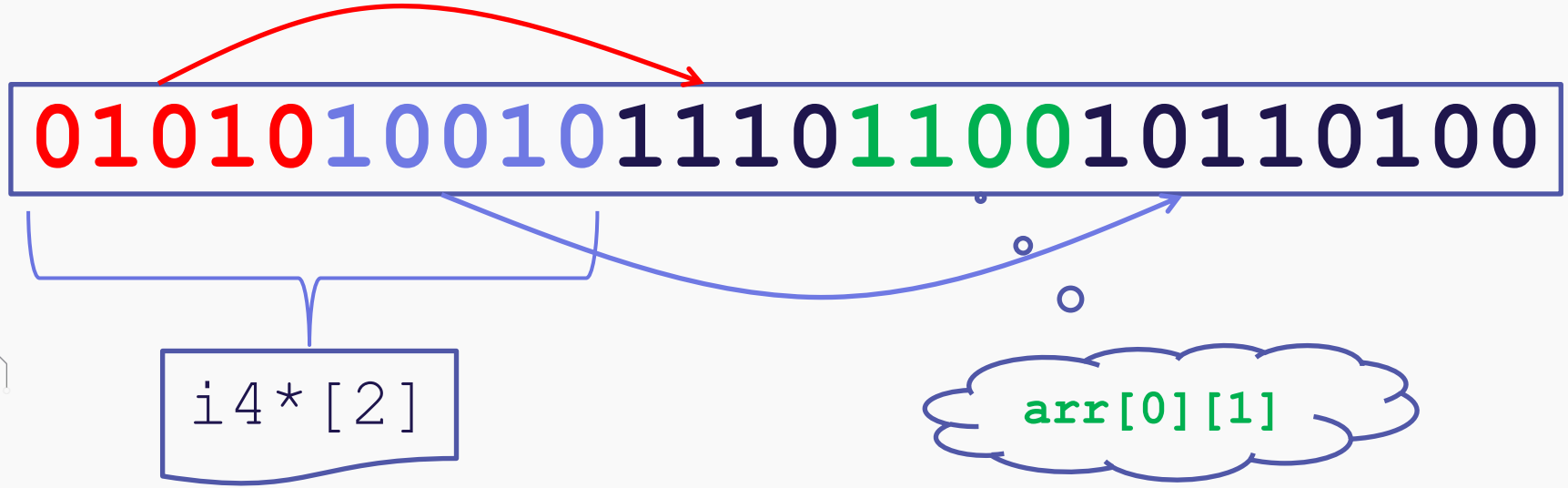
ОБСУЖДЕНИЕ

Кто-нибудь понимает, **почему** row-major?

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

ДВУМЕРНЫЕ МАССИВЫ: JAGGED ARRAYS

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

Интересный вопрос: как во всех четырех случаях вычисляется доступ к соответствующему элементу?

ВЫЧИСЛЕНИЕ АДРЕСОВ

- Массиво-подобное вычисление

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

ОПУСКАНИЕ ИНДЕКСОВ

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}}; // ?
```

ОПУСКАНИЕ ИНДЕКСОВ

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[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)];
```

НА БУДУЩЕЕ: 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?
- Вам придется подумать об умножении матриц и оптимизациях при выборе любого из этих подходов.
- Подумайте о других матричных операциях: перестановка строк местами или транспонирование.
- При выборе любого из способов решения будут довольно сильно различаться

КОНСТРУКТОРЫ

```
class Matrix {
    // некое представление
public:
    // конструктор для создания матрицы, заполненной значением
    Matrix(int cols, int rows, int val = 0);

    //конструктор для создания заданной последовательности
    Matrix(int cols, int rows, It start, It fin);
};
```

Как мне написать конструктор, который создает единичную матрицу? А диагональную? А треугольную?

ВСПОМНИМ

Что является инвариантом RAII класса?

СТАТИЧЕСКИЕ МЕТОДЫ И ДРУЗЬЯ

Кроме методов класса доступ к закрытой части есть у статических и дружественных функций

```
class C {
    int x = 0;
public:
    int get_x() const { return x; }
    static int s_get_x(const S *s) { return s->x; }
    friend int f_get_x(const S *s);
};

int f_get_x(const S *s) { return s->x; }
```

ДИАГРАММА ВОЗМОЖНОСТЕЙ

	Методы	Статические функции	Друзья
Получает неявный указатель this	Да	Нет	Нет
Находится в пространстве имен класса	Да	Да	Нет
Имеет доступ к закрытому состоянию класса	Да	Да	Да

ДРУЖБА – ЭТО МАГИЯ

- Статические функции более безопасны. Они являются частью интерфейса класса и их пишет разработчик, который заботится о сохранении инвариантов.
- Функции-друзья обычно пишет кто-то другой, и они могут нарушать инвариант класса.
- Особенно опасно дружить с целыми классами.

- В целом, дружба часто связана с магией и, заводя друзей, вы почти всегда ошибаетесь.

- Общее правило можно сформулировать так:

Иметь много друзей – ошибка. И давать друзьям доступ к приватной области – может быть опасно!

КОНСТРУКТОРЫ

```
class Matrix {
    // некое представление
public:
    // конструктор для создания матрицы, заполненной
значением
    Matrix(int cols, int rows, int val = 0);

    //конструктор для создания заданной последовательности
    Matrix(int cols, int rows, It start, It fin);

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

БОЛЬШАЯ ПЯТЕРКА

```
class Matrix {
    // некое представление
public:
    // копирующий и перемещающий конструктор
    Matrix(const Matrix& rhs);
    Matrix(Matrix&& rhs);
    // присваивание и перемещение
    Matrix& operator=(const Matrix& rhs);
    Matrix& operator=(Matrix&& rhs);
    // деструктор
    ~Matrix();
};
```

СПОЙЛЕР: АННОТАЦИЯ NOEXCEPT

- Если вы уверены, что ваш метод делает только примитивные операции над примитивными типами (например, обменивает указатели и только), вы можете аннотировать его как noexcept
- Мы пока не очень понимаем детали этого, но похоже мы можем так пометить перемещающие конструкторы и операторы присваивания

```
Matrix(Matrix&& rhs) noexcept;
Matrix& operator=(Matrix&& rhs) noexcept;
```

- Пока что вешайте эту аннотацию очень осторожно и только там, где вы уверены, что вы не врете. Любое копирование обобщенного типа блокирует это.

СЕЛЕКТОРЫ

```
class Matrix {
    // некое представление
public:
    // базовые
    int ncols() const;
    int nrows() const;

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

УДОБНЫЕ МЕТОДЫ

```
class Matrix {
    // некое представление
public:
    // отрицание
    Matrix& negate() &;

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

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

Как сделать доступ к этим элементам?

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

УДОБНЫЕ МЕТОДЫ

```
class Matrix {
    // некое представление
public:
    // отрицание и транспонирование
    Matrix& negate() &;
    Matrix& transpose() &;

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

    // доступ к элементам
    ??? operator[](int x) const; // что он возвращает?
};
```

PROXY-ОБЪЕКТЫ

```
class Matrix {
    // некое представление
    struct ProxyRow {
        double *row;
        double& operator[](int n) { return row[n]; }
        const double& operator[](int n) const { return row[n]; }
};
public:
    // мы хотим использовать m[x][y]
    ProxyRow operator[](int);
};
```

Запомним идею создания прокси-объектов. Она нам еще не раз спасет жизнь.

ОБСУЖДЕНИЕ

Мы научились переопределять несколько основных операторов:

- Приведение
- Присваивание
- Разыменовывание
- Стрелочка
- Индексаторы

Все они могут быть **только** методами

Но вообще над нашими матрицами и иными вашими объектами возможны другие операции, например сложение и умножение на константу.

Скоро мы научимся переопределять все возможные операторы.

ИДЕЯ UNIQUE_PTR

Основная идея – использовать для передачи управления перемещение:

```
unique_ptr(const unique_ptr& rhs) = delete;

unique_ptr(unique_ptr&& rhs) : ptr_(rhs.ptr_) {
    rhs.ptr = nullptr; }

unique_ptr& operator=(unique_ptr&& rhs) {
    swap(*this, rhs); return *this;
}
```

Это очень удобный класс, позволяющий вам не писать свои велосипеды

ПЕРЕДАЧА ЗА SCOPE

Уникальное владение можно передать в другой scope:

```
int foo(int x, double y) {
    std::unique_ptr<MyRes> res{new MyRes(x, y)}; // захват
    ….
    if (condition) {
        bar(std::move(res)); // корректная передача владения
        return 1;
    }
    ….
    return 0; // освобождается в деструкторе
}
```

Теперь `bar()` принимает `unique_ptr`, который не может быть скопирован.

УДОБНОЕ СОЗДАНИЕ

Пока что это выглядит, как волшебство

```
int foo(int x, double y) {
    auto res = std::make_unique<MyRes>(x, y); // захват
    ….
    if (condition) {
        bar(std::move(res)); // корректная передача владения
        return 1;
    }
    ….
    return 0; // освобождается в деструкторе
}
```

Вы научитесь творить такое волшебство, если поступите в ВУЗ и продолжите изучать прекрасный C++

ВОПРОС НА ПОДУМАТЬ

Что вы скажете об этом?

```
const std::unique_ptr<MyRes> p{new MyRes(x, y)};
```

ВОПРОС НА ПОДУМАТЬ

Что вы скажете об этом?

```
const std::unique_ptr<MyRes> p{new MyRes(x, y)};
```

Тот случай, когда const по сути создает принципиально новый объект:

- Его нельзя скопировать, потому что это `unique_ptr`
- Его нельзя передать, потому что он `const`

ТИПЫ ГОРАЗДО ВАЖНЕЕ В C++ ЧЕМ В C

В заголовке изложено неоспоримое утверждение!

- Типы участвуют в разрешении имён
- Типы могут иметь ассоциированное поведение
- За счет шаблонной параметризации типов может быть куда больше, их куда проще порождать из обобщенного кода

Но при этом всем, любой объект – лишь кусок памяти.

```
float f = 1.0;
char x = *((char *)&f + 2); // это легально. Что в x?
```

ОБСУЖДЕНИЕ

Не имеет ли приведение в стиле C каких-то тёмных сторон и подвохов?

ОБСУЖДЕНИЕ

Не имеет ли приведение в стиле C каких-то тёмных сторон и подвохов?

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 C всё пишется одинаково

ПРИВЕДЕНИЯ В СТИЛЕ C++

- `static_cast` – это 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` – это 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 и это хорошо.

НЕМНОГО О C++20

Побитовая реинтепретация 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 влипнуть в C-style cast

Итак, почти всегда наш выбор – это `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<T>(S)); // OK
```

То же самое касается и синтаксиса копирующей инициализации

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

РЕЗЮМИРУЕМ

- Кроме того, что C++ style casts позволяют чётко указать, что вы хотите, они еще 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 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 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;
// 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 из вас исчерпывающе изложить сишную часть правил с первого семестра?

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, 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); // foo(Foo);
foo(+f); // foo(int);
```

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Edsger W. Dijkstra – Go To Statement Considered Harmful, 1968
2. Edsger W. Dijkstra – The Humble Programmer, ACM Turing Lecture, 1972
3. Скотт Мейерс, Эффективный современный C++: 42 способа улучшить ваше использование C++11 и C++14
4. Klaus Iglberger – Back to Basics: Move Semantics, CppCon, 2019
5. RAII and Rule of Zero, Arthur O'Dwyer, CppCon, 2019
6. Nicolai Josuttis – The Nightmare of Move Semantics for Trivial Classes, 2017
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.