RAII И ПЕРЕМЕЩЕНИЕ

Владение ресурсом и семантика перемещения. Кроме того немного правды о двумерных массивах

К. Владимиров, Intel, 2021
mail-to: konstantin.vladimirov@gmail.com

- Владение ресурсом и RAII
- Семантика перемещения
- Двумерные массивы
- Проектирование матрицы

Владение ресурсом

- Памятью владеет тот, кто её выделяет и освобождает.

```
S *p = new S;
foo(p); // foo(S*);
delete p;
```

- Что может пойти не так в этом коде?

Владение ресурсом

- Памятью владеет тот, кто её выделяет и освобождает.

```
S *p = new S;
foo(p); // foo(S *p) { delete p; }
delete p;
```

- Что может пойти не так в этом коде?
- В общем случае память это только один из возможных ресурсов.

Забавный пример

```
template <typename S> int foo(int n) {
  S *p = new S{n};
  // .... some code ....
  if (condition) {
    delete p;
    return FAILURE;
  }
  // .... some code ....
  delete p;
  return SUCCESS;
}
```

- Хотелось бы иметь одну точку освобождения чтобы избежать проблем

Страшное goto

```
template <typename S> int foo(int n) {
  S *p = new S{n}; int result = SUCCESS;
  // .... some code ....
  if (condition) {
    result = FAILURE;
    goto cleanup;
  }
  // .... some code ....
cleanup:
  delete p;
  return result;
}
```


Linux kernel coding style

Социально-приемлимое goto

```
template <typename S> int foo(int n) {
  S *p = new S{n}; int result = SUCCESS;
  do {
    // .... some code ....
    if (condition) {
      result = FAILURE;
      break;
    }
    // .... some code ....
  } while(0);

  delete p;
  return result;
}
```

Отступление: goto considered harmful

- Что вы думаете о вот таком коде?

```
struct X {
  int smth = 42;
};

int foo(int cond) {
  switch(cond) {
  case 0: X x;
  case 1: return x.smth; // 42?
  }
}
```

Отступление: goto considered harmful

- Что вы думаете о вот таком коде?

```
struct X {
  int smth = 42;
};

int foo(int cond) {
  switch(cond) {
  case 0: X x;
  case 1: return x.smth; // FAIL
  }
}
```

- К счастью это ошибка компиляции

Обсуждение

- Какие мы знаем goto-маскирующие конструкции?

`switch-case`, `break`, `continue`, `return`, ещё?

- Будьте со всеми ними крайне осторожны при работе с конструкторами и деструкторами. Ваш выбор явные блоки

```
int foo(int cond) {
  switch(cond) {
  case 0: { X x; }
  case 1: return x.smth; // очевидная ошибка, x не виден
  }
}
```

RAII: resource acquisition is initialization

- Чтобы не писать goto можно спроектировать класс, в котором конструктор захватывает владение, а деструктор освобождает ресурс

```
template <typename S> int foo (int n) {
  ScopedPointer<S> p{new S(n)}; // ownership passed
  // .... some code ....
  if (condition)
    return FAILURE; // dtor called: delete
  // .... some code ....
  return SUCCESS; // dtor called: delete
}
```

- Как этот класс мог бы выглядеть?

RAII обёртка

- Как мог бы выглядеть упомянутый ScopedPointer?

```
template <typename T> class ScopedPointer {
  T *ptr_;

public:
  ScopedPointer(T *ptr = nullptr) : ptr_(ptr) {}
  ~ScopedPointer() { delete ptr_; }
```

- И у нас есть две проблемы. Первая: как написать копирование/присваивание
- Вторая: как сделать с ним что-то полезное, не дав утечь указателю?

Глубокое копирование

- Начнем с копирования

```
template <typename T> class ScopedPointer {
  T *ptr_;

public:
  ScopedPointer(T *ptr = nullptr) : ptr_(ptr) {}
  ~ScopedPointer() { delete ptr_; }

  ScopedPointer(const ScopedPointer& rhs) :
    ptr_(new T{*rhs.ptr_}) {}

  // как бы вы реализовали присваивание?
  ScopedPointer& operator= (const ScopedPointer& rhs);
```

Доступ к состоянию

- Можно сделать просто функцию вроде access

```
template <typename T> class ScopedPointer {
  T *ptr_;

public:
  ScopedPointer(T *ptr = nullptr) : ptr_(ptr) {}
  ~ScopedPointer() { delete ptr_; }

  T& access() { return *ptr_; }
  const T& access() const { return *ptr_; }
```

- Итог немного многословен

```
ScopedPointer<S> p{new S(n)}; int x = p.access().x; // (*p).x
```

Перегрузка разыменования

- Разыменование указателя это оператор и он перегружается

```
template <typename T> class ScopedPointer {
  T *ptr_;

public:
  ScopedPointer(T *ptr = nullptr) : ptr_(ptr) {}
  ~ScopedPointer() { delete ptr_; }

  T& operator*() { return *ptr_; }
  const T& operator*() const { return *ptr_; }
```

- Уже сейчас стало гораздо лучше, но хотелось бы, конечно, стрелочку

```
ScopedPointer<S> p{new S(n)}; int x = (*p).x; // p->x
```

Проблема со стрелочкой

```
template <typename T> class ScopedPointer {
  T *ptr_;

public:
  T& operator*() { return *ptr_; }
  const T& operator*() const { return *ptr_; }

  ??? operator->() { return ???; } // например p->x, для T::x
```

- А что собственно возвращать?

Решение: drill down

```
template <typename T> class ScopedPointer {
  T *ptr_;

public:
  T& operator*() { return *ptr_; }
  const T& operator*() const { return *ptr_; }

  T* operator->() { return ptr_; } // например p->x, для T::x
  const T* operator->() const { return ptr_; }
```

- Вызов `p->x` эквивалентен `(p.operator->())->x` и так сколько угодно раз
- Стрелочка как бы "зарывается" в глубину на столько уровней на сколько может

drilldown.cc

Обсуждение

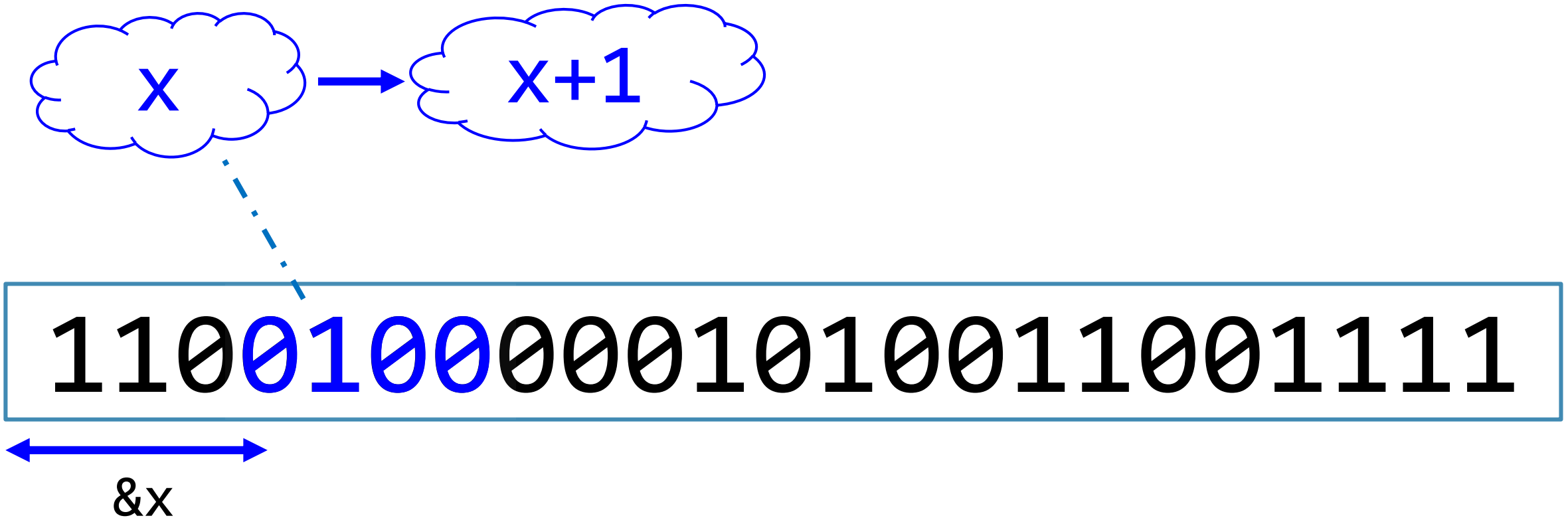
- Хорош ли получившийся scoped pointer? Подумайте вот о чём.

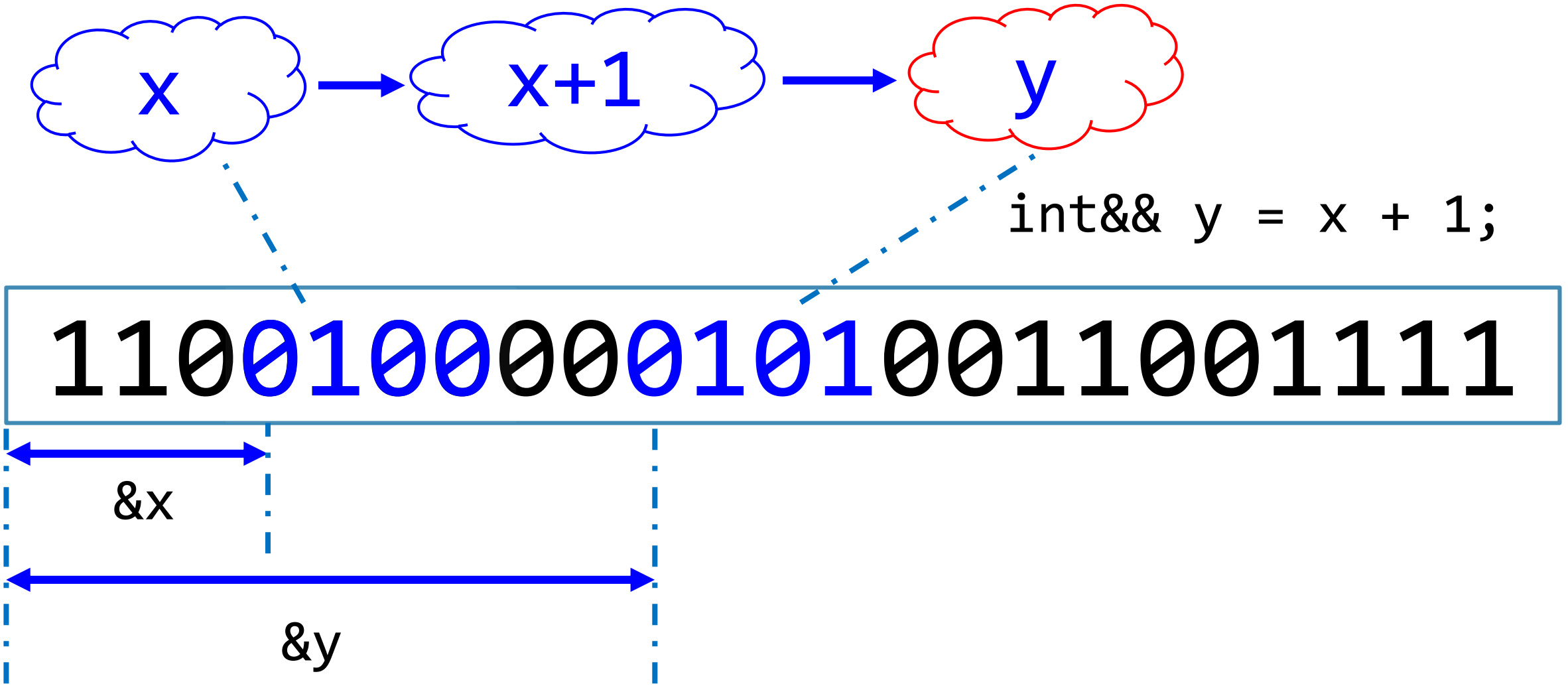
```
S *a = new S(1), *b = new S(2);
std::swap(a, b); // что происходит тут?

ScopedPointer<S> x{new S(1)}, y{new S(2)};
std::swap(x, y); // а что тут?
```

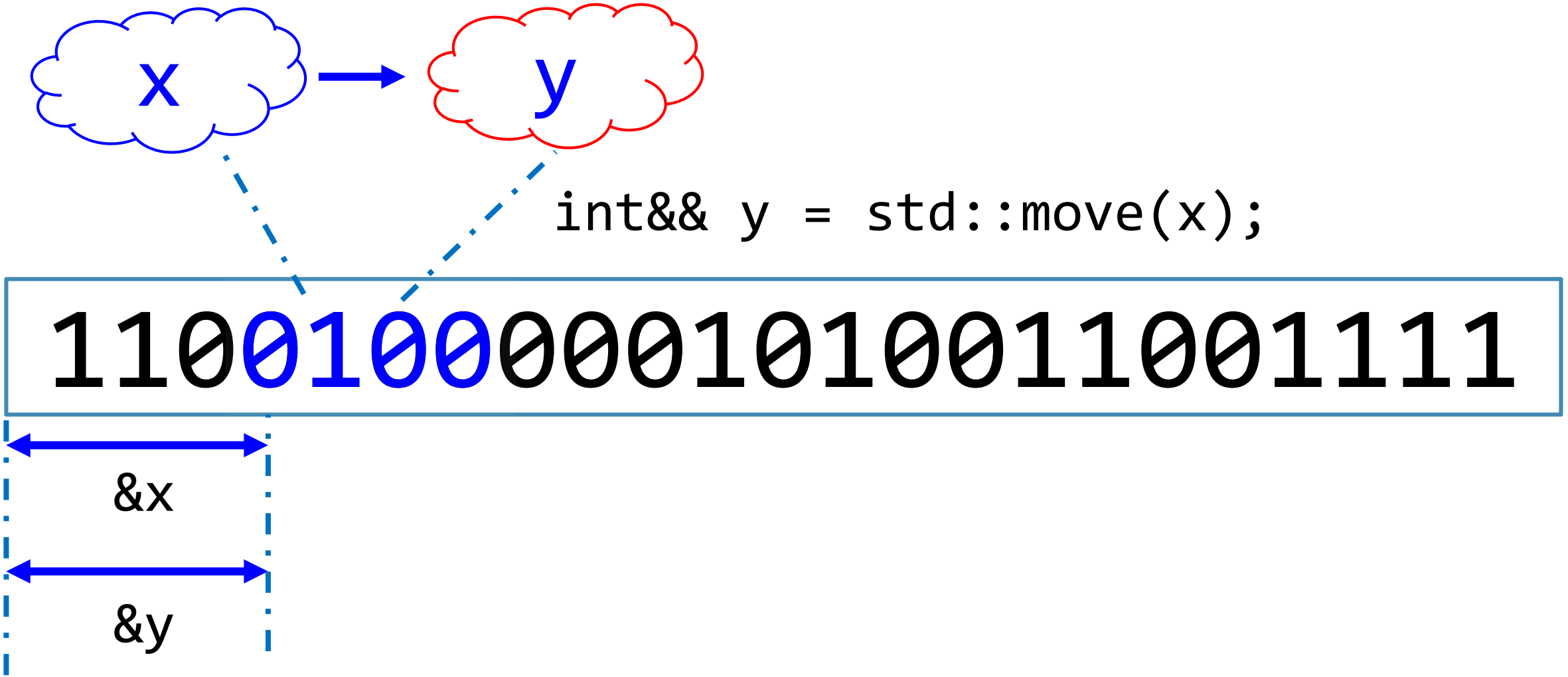
- Для справки: `std::swap` в C++98 был определен так:

```
template <typename T> void swap (T& x, T& y) {
  T tmp = x; // copy ctor
  x = y;     // assign
  y = tmp;   // assign
}
```

- Владение ресурсом и RAII
  - Семантика перемещения
- Двумерные массивы
- Проектирование матрицы


x
x+1
11001000001010011001111
&x


x
x+1
y
int&& y = x + 1;
1100100000010100111001111
&x
&y


x
y
int&& y = std::move(x);
1100100000101001100111
&x
&y

Кросс-связывание

- Правая ссылка не может быть связана с lvalue

```
int x = 1;
int &&y = x + 1; // ok
int &&b = x;     // fail, не rvalue
```

- Неконстантная левая ссылка не может быть связана с rvalue

```
int &c = x + 1;         // fail, не lvalue
const int &d = x + 1; // ok, продляет время жизни
```

- Но при этом правая ссылка сама по себе задаёт имя и адрес и является lvalue

```
int &&e = y; // fail, не rvalue
int &f = y;  // ok
```

Обсуждение: методы над rvalues

- Всегда надо помнить: метод может быть вызван для rvalue-expression

```
struct S {
  int n = 0;
  int& access() { return n; }
};

S x;
int& y = x.access(); // ok

int& z = S{}.access(); // ok?
```

Обсуждение: методы над rvalues

- Всегда надо помнить: метод может быть вызван для rvalue-expression

```
struct S {
  int n = 0;
  int& access() { return n; }
};

S x;
int& y = x.access(); // ok

int& z = S{}.access(); // this parrot is dead
```

Аннотация методов

- Методы могут быть аннотированы и различать lvalue от rvalue

```
struct S {
  int foo() &; // 1
  int foo() &&; // 2
};

extern S bar ();

S x {};

x.foo(); // 1
bar().foo(); // 2
```

Аннотация методов

- Аннотация методов крайне полезна

```
struct S {
  int n = 0;
  int& access() & { return n; }
};

S x;
int& y = x.access(); // ok

int& z = S{}.access(); // ошибка компиляции
```

- Ошибка компиляции всегда лучше, чем висячая ссылка

Аннотация методов

```
class X {
  vector<char> data_;
public:
  X() = default;
  vector<char> const & data() const & { return data_; }
  vector<char> && data() && { return std::move(data_); }
};

X obj;
vector<char> a = obj.data(); // copy
vector<char> b = X().data(); // move
```

- Здесь есть некая экономия на спичках и нет висячей правой ссылки
- Это не значит, что нужно бросаться так делать, разумеется

Аккуратнее с возвратом правых ссылок

- Возврат правых ссылок часто ведет себя плохо

```
int& foo(int& x) { return x; } // ok
const int& bar(const int& x) { return x; } // когда как

int&& buz(int&& x) { return std::move(x); } // DANGLE
```

- Обычно вы не хотите их возвращать если у вас не &&-аннотированный метод
- При этом:

```
int& bat(int&& x) { return x; } // снова когда как
```

- Так что правые ссылки с точки зрения провисания даже опаснее левых

Перемещающие конструкторы

- Конструктор берущий rvalue ref не обязан сохранять значение (т.к. это rvalue)
- Это потрясающе выгодно там, где требуется глубокое копирование

```
template <typename T> class ScopedPointer {
  T *ptr_;

public:
  ScopedPointer(const ScopedPointer& rhs) :
    ptr_(new T{*rhs.ptr_}) {}

  ScopedPointer(ScopedPointer&& rhs) : ptr_(rhs.ptr_) {
    rhs.ptr_ = nullptr;
  }
};
```

Перемещающее присваивание

- Для перемещающего присваивания есть варианты

```
ScopedPointer& operator= (ScopedPointer&& rhs) {
  if (this == &rhs)
    return *this;

  // вариант #1: оставляем пустое состояние
  delete ptr_;
  ptr_ = rhs.ptr_;
  rhs.ptr_ = nullptr;
  return *this;
}
```

- Оно обязано оставить объект в консистентном состоянии

Перемещающее присваивание

- Для перемещающего присваивания есть варианты

```
ScopedPointer& operator= (ScopedPointer&& rhs) {
  if (this == &rhs)
    return *this;

  // вариант #2: делаем обмен и пусть деструктор удаляет
  std::swap(ptr_, rhs.ptr_);
  return *this;
}
```

- Это состояние, вообще говоря, не обязано быть предсказуемым

Эффективный обмен значениями

- Старый способ обмена значениями

```
template <typename T> void swap (T& x, T& y) {
  T tmp = x; // copy ctor
  x = y;     // assign
  y = tmp;   // assign
}
```

- Новый способ очевидно лучше

```
template <typename T> void swap (T& x, T& y) {
  T tmp = std::move(x); // move ctor
  x = std::move(y);     // move assign
  y = std::move(tmp);   // move assign
}
```

Аккуратнее с move on result

- Обычно в таком коде std::move просто не нужен

```
T foo(some arguments) {
  T x = some expression;
  // more code
  return std::move(x); // не ошибка, но зачем?
}
```

- Функция, возвращающая by value это rvalue expression и таким образом всё равно делает move в точке вызова.
- При этом использование std::move может сделать вещи чуть хуже, убив RVO
- Ограничьте move on result случаями возврата ссылки

Задача: особенности move

```
int x = 1;
int a = std::move(x);
assert (x == a); // ???

ScopedPointer y {new int(10)};
ScopedPointer b = std::move(y);
assert (y == b); // ???
```

- Что можно сказать о приведённых assertions?

Решение: особенности move

```
int x = 1;
int a = std::move(x);
assert (x == a); // всегда выполнено

ScopedPointer y{new int(10)};
ScopedPointer b = std::move(y);
assert (y == b); // мы не знаем
```

- Использование move всего лишь получает && ничего не делая с переменной.
- Будет ли состояние потеряно зависит от того есть ли у класса перемещающий конструктор и как он реализован.
- У int его точно нет, на чём и построен первый ответ.

Проблема implicit move

- Перемещение по умолчанию перемещает по умолчанию все поля класса.

```
template <typename T> class SillyPointer {
  T *ptr_;

public:
  SillyPointer(T *ptr = nullptr) : ptr_(ptr) {}
  ~SillyPointer() { delete ptr_; }
};

template <typename T> void swap(T& lhs, T& rhs) {
  T tmp = std::move(lhs);
  lhs = std::move(rhs);
  rhs = std::move(tmp);
} // UB (probably segfault)
```

Правило пяти

- Классическая идиома проектирования rule of five утверждает, что:

"Если ваш класс требует нетривиального определения хотя бы одного из пяти методов:

1. *копирующего конструктора*
2. *копирующего присваивания,*
3. *перемещающего конструктора*
4. *перемещающего присваивания*
5. *деструктора*

то вам лучше бы нетривиально определить все пять"

- Очевидно SillyPointer его нарушает: он определяет нетривиальный деструктор и только его.

Краевой случай: move from const

- Итак, хорошо организованный move ctor изменяет rhs. Но что если rhs нельзя изменить?

```
const ScopedPointer<int> y{new int(10)};

ScopedPointer<int> b = std::move(y); // копирование
```

- В этом случае move ctor просто не будет вызван, так как его сигнатура предполагает Buffer&& а не Buffer const &&.
- Вместо этого, Buffer const && будет приведён к Buffer const & и вызовется копирующий конструктор, несмотря на явное указание move.

Правило нуля

- Классическая идиома проектирования rule of zero утверждает, что:

"Если ваш класс требует нетривиального определения хотя бы одного из пяти неявных методов, и, таким образом, все пять

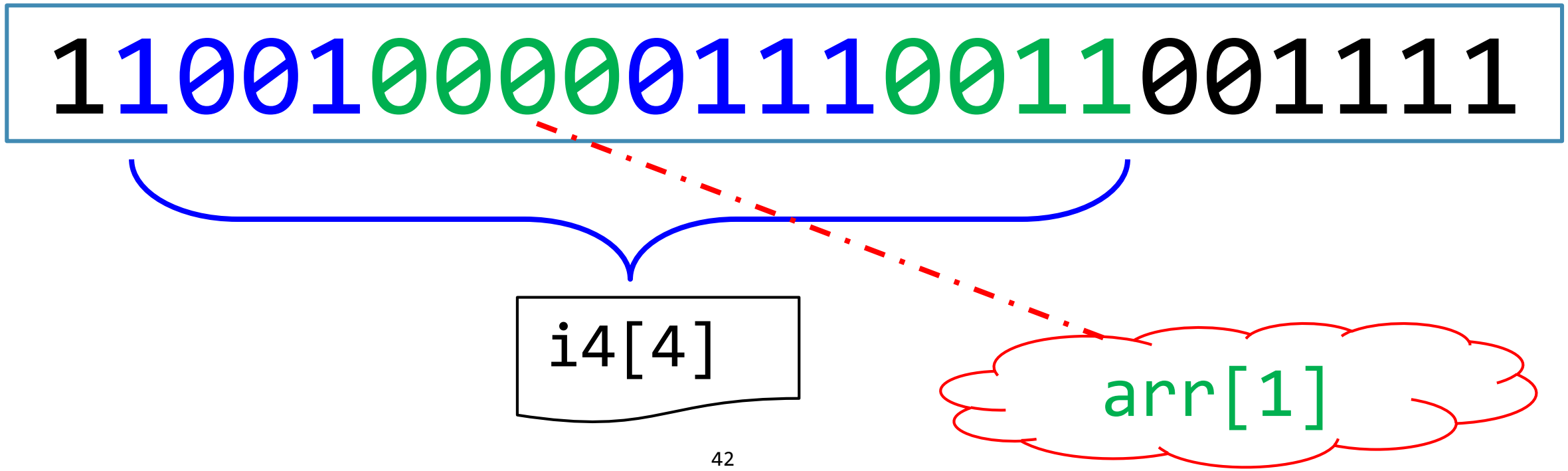
То в нём не должно быть никаких других методов"

- Это моё любимое правило. Оно нас ещё много раз выручит.

- Владение ресурсом и RAII
- Семантика перемещения
- Двумерные массивы
- Проектирование матрицы

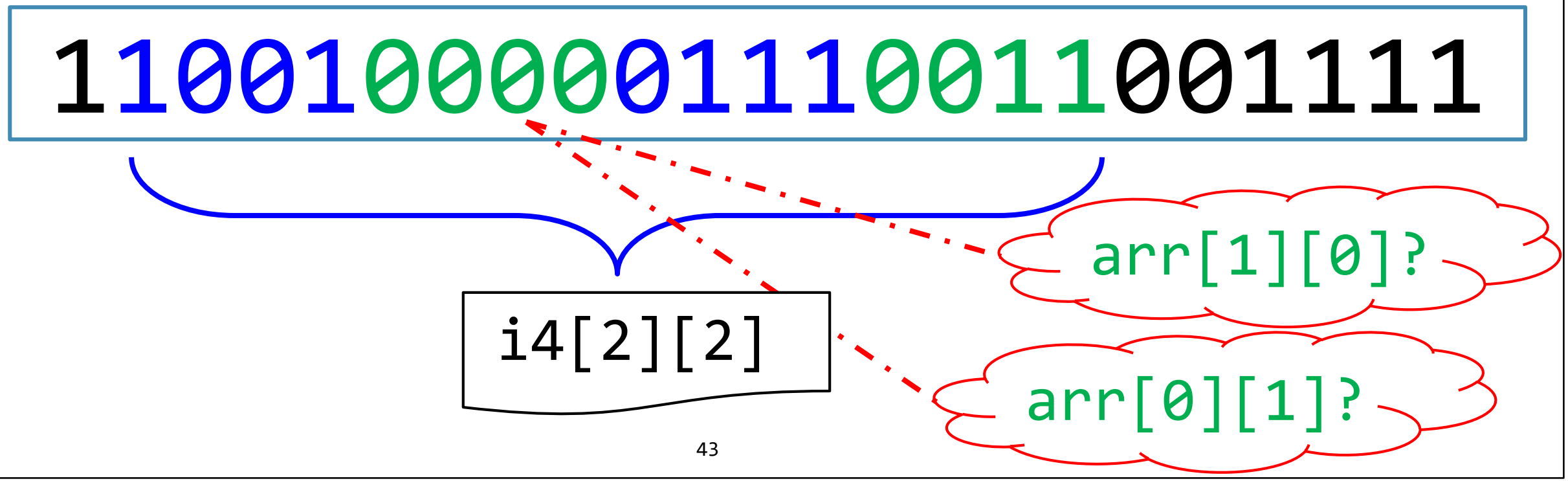
Двумерные массивы

- 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

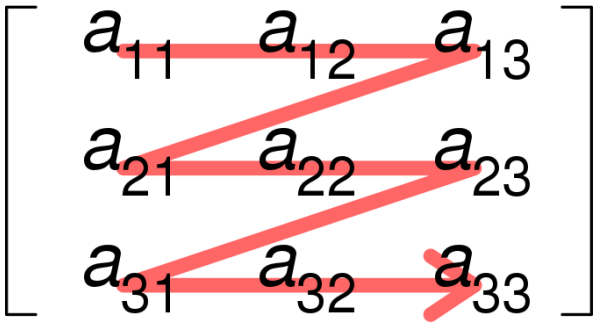


Row-major vs column-major

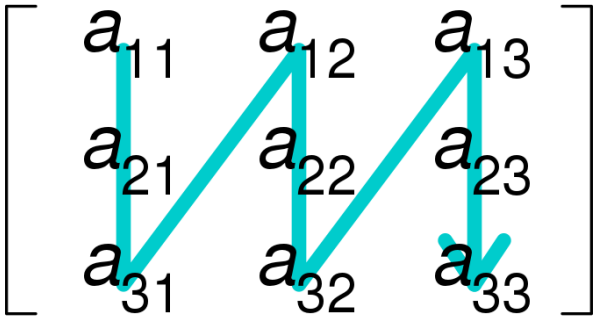
- В математике для матрицы $\{a_{ij}\}$, первый индекс называется индексом строки, второй – индексом столбца
- В языке C принят row-major order (очень просто запомнить: язык C читает матрицы как книжки)
- row-major означает, что первым изменяется самый внешний индекс

```
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

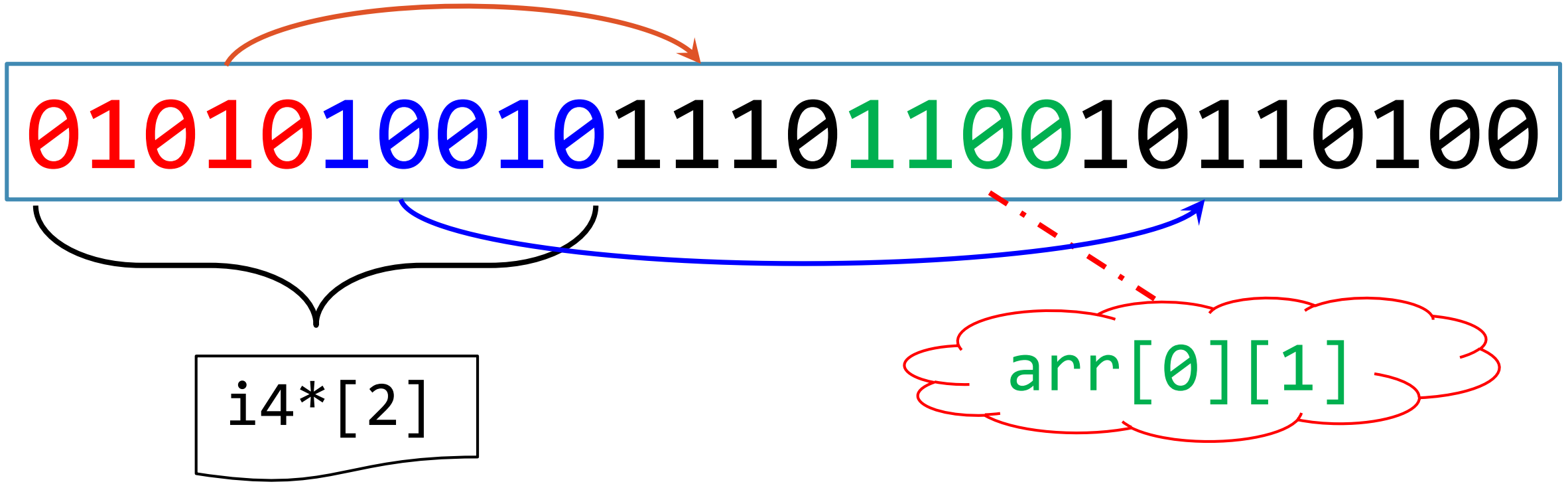
- Кстати, а кто-нибудь понимает **почему** row-major?

```
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

- Владение ресурсом и RAII
- Семантика перемещения
- Двумерные массивы
- Проектирование матрицы

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);
```

- Как мне написать конструктор для создания единичной матрицы?

Обсуждение

- Что является инвариантом любого RAII класса?

Статические методы и друзья

- Кроме методов класса, доступ есть у статических и дружественных функций

```
class S {
  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 на this	да	нет	нет
находится в пространстве имён класса	да	да	нет
имеет доступ к закрытому состоянию класса	да	да	да

Обсуждение: 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 инварианты как хотят
- Особенно опасно дружить с целыми классами
- В целом дружба часто бывает связана с магией и заводя себе друзей вы почти всегда ошибаетесь

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 пятёрка

```
template <typename T> 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();
```

Тизер: 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.

```
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 врёте. Любое копирование обобщённого T блокирует это.

Селекторы

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

public:
 // базовые
 int ncols() const;
 int nrows() const;

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

Удобные методы

```
template <typename T> 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 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

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

public:
  // отрицание и транспонирование
  Matrix& negate() &;
  Matrix& transpose() &;

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

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

Прокси-объекты

```
template <typename T> class Matrix {
  // ..... тут некое представление .....
  struct ProxyRow {
    T *row;
    const T& operator[](int n) const { return row[n]; }
    T& operator[](int n) { return row[n]; }
  };
public:
 // Мы бы хотели использовать m[x][y]
 ProxyRow operator[](int);
```

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`

Обсуждение

- Мы научились переопределять несколько основных операторов:
  - приведение
  - присваивание
  - разыменование
  - стрелочка
  - индексаторы
- Все они могут быть только методами
- Но вообще над нашими матрицами возможны другие операции, например сложение или умножение на константу
- Скоро мы научимся переопределять все возможные операторы

ДОПОЛНИТЕЛЬНО

Использование unqiue_ptr

Идея unique_ptr

- Основная идея: использовать для передачи управления перемещение

```
unique_ptr(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 (какое-то условие) {
    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 выглядит немного волшебством

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

- Мы научимся писать такие вещи во второй части курса

Обсуждение

- Что вы скажете о таком подходе?

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

Литература

- [CC11] ISO/IEC 14882 – "Information technology – Programming languages – C++", 2011
- [BS] Bjarne Stroustrup – The C++ Programming Language (4th Edition), 2013
- [ED] Edsger W. Dijkstra – Go To Statement Considered Harmful, 1968
- [EDH] Edsger W. Dijkstra – The Humble Programmer, ACM Turing Lecture, 1972
- [SM] Scott Meyers, "Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14"
- [KI] Klaus Iglberger – Back to Basics: Move Semantics, CppCon, 2019
- [AD] RAII and the Rule of Zero - Arthur O'Dwyer, CppCon, 2019
- [NJ] Nicolai Josuttis – The Nightmare of Move Semantics for Trivial Classes, 2017
- [KVR] Kris van Rens – Understanding value categories in C++, C++ Dublin User Group, 2020

СЕКРЕТНЫЙ УРОВЕНЬ

Правло для prvalue elision

Новое правило: prvalue elision

- Начиная с 2017, компилятор обязан делать copy/move elision для `prvalues`

```
class NoCopyMove {
 NoCopyMove() = default;
 NoCopyMove(const NoCopyMove&) = delete;
 NoCopyMove(NoCopyMove&&) = delete;
};

void foo(NoCopyMove x);

foo(NoCopyMove{}); // легально в C++17
```

- Во многих компиляторах это работало и раньше
- Обратите внимание: стандарт форсит это только для `prvalues`

Новое правило: prvalue elision

- Не надо путать с NRVO

```
NoCopyMove bar() { return NoCopyMove{}; } // легально в C++17

NoCopyMove n = bar(); // легально в C++17

NoCopyMove buz() {
  NoCopyMove t;
  return t; // всё ещё зависит от доброй воли компилятора
}
```

- Если эта конструкция вообще легальна, компилятор её дальше соптимизирует