Оглавление

АРГУМЕНТЫ ПО УМОЛЧАНИЮ

```
     //сначала просто  // потом по умолчанию (… аргументы – времени компиляции!)
void foo(int a, int b, int c=3, int d=2) {…} // Вот, что сделает компилятор:
```

```
void foo(int a, int b, int c, int d)  {…}
inline void foo(int a, int b, int c)  { foo(a, b, c, 2); }
inline void foo(int a, int b)         { foo(a, b, 3); }
```

```
foo(1, 1, 2); // те, что по умолчанию, можно не писать

// Последовательность присвоений в аргументах в стандарте не определена, поэтому…
void fun(int a, int b = a, int c = b) {…} // ОШИБКА т.к. возможны 2 сценария:
  1) сначала b = значение из a, после чего c = значение из b (т.е. значение из a)
  2) сначала c = значение из b (т.е. мусор), после чего b = значение из a
// Поэтому нужно явно писать самостоятельно:
```

```
void foo(int a, int b, int c)   {…}
inline void foo(int a, int b)   { foo(a, b, c); }
inline void foo(int a)          { foo(a, a); }
```

```
foo(3); // сюда будет встроена: foo(3, 3); // сюда будет встроена: foo(3, 3, 3);
```

СТАНДАРТНЫЕ ТИПЫ (из std)

```
nullptr_t // - тип данных, который может иметь только одно значение:
     nullptr //неявно приводится к нулевому указателю на соответствующий тип type*
size_t // - тип возвращаемого значения оператора sizeof(…);
     Размер типа: такой, чтобы можно было записать размер любого массива т.е.:
          в 32-битной системе sizeof(size_t); // => 4 т.е. 4 byte т.е. 32 bits
          в 64-битной системе sizeof(size_t); // => 8 т.е. 8 byte т.е. 64 bits
     Таким образом в переменную типа size_t может быть безопасно помещён указатель
     Диапазон: от 0 до SIZE_MAX (т.е. он беззнаковый)
```

ВЫРАЖЕНИЯ

ИНКРЕМЕНТ

```
a++; // создастся временная = a, потом изменение a, потом возврат временной
++a; // изменение a, потом возврат результата(работает быстрее т.к. без временной)
                ++a идентично a+=1 похоже на a=a+1
```

ПОБИТОВЫЕ ОПЕРАЦИИ

```
5|6;  // ИЛИ      => 101 | 110 => 111 => 7
5&6;  // И       => 101 & 110 => 100 => 4
5^6;  // XOR (исключающее или) => 101 ^ 110 => 011 => 3
5<<3; // СДВИГ на 3 влево  => 101 => 101000 => 40 (т.е. *2^3)
5>>1; // СДВИГ на 3 вправо => 101 => 10 => 2
// но если слева от << или >> стоит объект потока => будет вывод/ввод в поток
~5;   // ОТРИЦАНИЕ     => 0…0101 => 1…1010 => -6 (в десятичной)
      // т.к. доп. код => вычтем единицу 11…1001 => инверсия 10…0110 => -6
```

ЛОГИЧЕСКИЕ ОПЕРАТОРЫ

```
! условие; // в результате получим отрицание условия
//Оптимизация: если по первому условию всё понятно, то второе выполняться не будет
условие_1 && условие_2; // условие_1 вернуло false => условие_2 не выполняется
условие_1 || условие_2; // условие_1 вернуло true  => условие_2 не выполняется
```

```
for (size_t i = 0; i < 10; ++i) {
     if (i == 0 || v[i] != v[i - 1]) { /*...*/ }
} // на 1-ом шаге 2-е условие не проверяется => не будет отрицательного индекса
```

НЕЯВНЫЕ КОНВЕРСИИ ТИПОВ

```
5/2.0   // 2.5  (какой-то операнд double => результат double)
'a'+1   // 98   (оба операнда НЕ double => результат int)
'a'+'b' // 195  (оба операнда НЕ double => результат int)
     // ОСТОРОЖНО:
```

```
double temp_f = 9/5 * temp_c + 32; // 9/5 = 1, исправим: 9.0/5
```

```
a < b < c идентично (a < b) < c идентично (true или false) < c
```

СРАВНЕНИЕ ВЕЩЕСТВЕННЫХ ЧИСЕЛ

```
// double и float хранятся с некоторой точностью, из-за которой при арифметических
операциях могут накапливать ошибку, например 0.3*3 != 0.9, поэтому:
if (var_1 == var_2) // так сравнивать не стоит, нужно так:
```

```
bool is_equal(double x, double y) { // - только так сравниваем double x, y;
     return std::fabs(x - y) < 100 * DBL_EPSILON;
} // получился предикат фиксированной точности (100 * DBL_EPSILON)
```

УКАЗАТЕЛИ *

```
&a // взятие адреса у переменной a
int a = 5, *ptr_a = &a; // double * - указатель на тип double
     const const int const const * const const ptr_a = &a;
ptr_a           // - это (читаю справа налево) ptr… - так обозначают указатели
* const         // константный указатель на… (т.е. ptr_a менять нельзя)
const int const // константный тип int (т.е. значение *ptr_a менять нельзя)
cout <<*ptr_a <<" = "<< a << endl;//=> 6 = 6                т.к. * - разыменование
cout << ptr_a <<" = "<<&a << endl;//=> 00A2F878 = 00A2F878 т.к. & - взятие адреса
int **ptr_ptr_a = &ptr_a;// ptr_ptr_a – указатель на ptr_a, который указывает на a
int **ptr_ptr_a = &&a;//ОШИБКА т.к. адреса a – rvalue, а адрес от rvalue нельзя
++*ptr_a;  //разыменовал указатель, потом префиксный инкремент над разыменованным
(*ptr_a)++;//разыменовал указатель, потом постфиксный инкремент над разыменованным
*ptr_a++;  //постфиксный инкремент указателя (т.е. теперь он указывает после a)
             потом разыменовал нечто, что не является int… => UB
```

АРИФМЕТИКА УКАЗАТЕЛЕЙ

```
int *arr; // тип данных: указатель на int т.е. int*
arr+5;//сместить указатель на 5 размеров типа данных указателя т.е.: 5*sizeof(int)
*(arr+5); // разыменовать смещённый указатель => получим lvalue типа int
arr[5]; // взятие индекса идентично: *(arr+5);
(arr+2)[3]; // идентично *(arr+2+3); // идентично *(arr+5); // идентично arr[5];
5[arr]; // идентично *(5+arr); // идентично *(arr+5); // идентично arr[5];
```

ССЫЛКИ &

```
int a = 5;
const int c = 3;
      int & a_ref;          // ОШИБКА т.к. сразу нужна инициализация
      int & a_ref = 3;      // ОШИБКА т.к. справа должно быть lvalue
      int & a_ref = a;      // справа lvalue
const int & a_ref = a или c или 3;    // справа lvalue или const lvalue или rvalue
      int&& a_ref = a + b; // справа rvalue(т.к. временный объект => можем менять)
      const const int const const & const const a_ref = a;
a_ref           // - это (читаю справа налево) …ref -так обозначают ссылки
& const         // компилятор воспримет, но практического смысла нет
const int const // константный тип int (т.е. значение aRef менять нельзя)
++a_ref; // инкрементирует тот объект, с которым связана ссылка
int* ptr_a = &a_ref;  ++*ptr_a; // ещё раз инкрементируется (через указатель)
int& val = arr[f(i)][f(j)]; // теперь будет удобно работать с элементом массива
```

LVALUE = RVALUE

```
lvalue (locator value)//объект, который занимает идентифицируемое место в памяти
rvalue (right value) //может стоять только справа от присваивания
                     //это всё что НЕ lvalue т.е. от него не получится взять адрес
      ПЕРЕМЕННЫЕ:
a = 3;      // a – переменная, 3 – литерал
a = b;      // a положит в себя значение из b (обычный оператор присваивание)
c = a + b; // c = temp – временное значение: значение (из a) + значение (из b)
3 = a; ИЛИ 3 = 3; ИЛИ (a + 3) = 3 // ОШИБКИ т.к. слева от равно rvalue
const int a = 3; // a – это const lvalue ведь мы можем &a
      УКАЗАТЕЛИ:
int *pa = &a     // &a => адрес (объекта a), итого: lvalue указатель на lvalue a
*pa = 3          // *pa                  =>   a = 3
*(&a) = 3        // *адрес (объекта a)   =>   a = 3
*(arr + 2) = 3   // *адрес (2-го элемента массива arr)  =>  arr[2] = 3
&3    // ОШИБКА т.к. взятие адреса от rvalue
&&a;  // ОШИБКА т.к.  => &(&a)   => &адрес (объекта a)
      ФУНКЦИИ:
val = get_rvalue(); // где int get_rvalue() { return 10; }
val = get_rvalue(); // где int get_rvalue() { int a = 0; return a; }
get_rvalue() = 3; // ОШИБКА т.к. слева от равно rvalue
get_lvalue() = 3; // где int& get_lvalue() { static int a = 0; return a; }
get_lv_bad() = 3; // где int& get_lv_bad() { int a = 0; return a; }
                  // скомпилируется, но UB т.к. висячая ссылка
third_el(arr) = 3;// где int& third_el(int* arr) { return arr[2]; }
```