# Операторы и выражения C++




February 14, 2022

# Операторы и выражения

Выражения определяют, какие действия применяются к объектам.
Выражения состоят из операторов и операндов.
Операнды:

- объекты;
- литералы;
- выражения.

```
z = b * c + 5
```

Здесь:
**5** — литерал,
**z**, **b**, **c** — имена объектов,
b∗c, b∗c+5, z=b∗c+5 — выражения.

# Классификация операторов

## Арность

**Арность** — количество операндов:

1. **унарные** — 1 операнд
   &a, c++, ∗p, ~x, !b
   (префиксные и постфиксные)

2. **бинарные** — 2 операнда
   a+b, **cout** << '\n', c = 0, a[n], f(x)

3. **тернарный** — 3 операнда
   (x>0) ? b : c

**Приоритет:** в одном подвыражении более приоритетные операторы выполняются раньше. (На самом деле строится дерево вычислений)

```
z = b * c + d.x;
//4   2   3  1
```

Порядок можно задать явно скобками:

```
z = b * ( c + d );
//3   2     1
```

Групп приоритетов — много (сейчас 18). В этом документе группы приоритетов будут разделяться такой линией:

---

На самом деле в стандарте нет строгого определения приоритета

# Классификация операторов

## Ассоциативность

**Ассоциативность:** в случае равных приоритетов задаёт направление вычисления операторов.

**Левоассоциативные**: выполняются слева направо:

```
a + b - c + d;
//1   2   3
cout << "String" << 5 << endl;
//   1           2    3
```

**Правоассоциативные**: выполняются справа налево:

```
a = b = d = 0;
//3   2   1
```

# Результат и побочное действие

**Результат**: значение, полученное в результате работы оператора. Используется для дальнейшего вычисления выражения или используется инструкциями.

**Побочное действие**: изменение значений операндов.

```
int a = 5, b = 2, c;
c = a + b;  // 7 – результат (+),
            // тип – int

c = 2;  //  2 – результат,
        // побочное действие – изменение 'c'

a = b = d = 0;
```

# Оператор :: 2,L,1

Оператор разрешения (определения) области видимости (бинарный, левоассоциативный). Scope resolution. Левый операнд — имя пользовательского типа или пространства имён, правый — имя “оттуда”.

```
std::cout << 5 << std::endl;
MyClass::n = 0;
```

**std** — имя пространства имён (namespace). **cout** — член пространства имён std. **MyClass** — имя класса. **n** — статический член класса MyClass.

```
int x = 3; // глобальная переменная
int main()
{
  int x = 2; // локальный x скрывает глобальный x
  cout << x << ' ' << ::x << endl; // :: − глобальный x
  {  int x = 5 + ::x;  } // :: − глобальный x
  return 0;
}
```

**obj.member**
**pobj->member**
Операторы доступа к члену объекта по:
**.** — имени объекта “obj” и имени члена “member”.
**->** — указателю на объект “pobj” и имени члена “member”.

```
struct Z {   // пользовательский тип Z
  int a, b; // члены данных
  void f(); // функция-член
};
Z x; // создали объект x типа Z
x.a = 0;
Z *px = &x; // указатель на объект
px->a = 9;
x.f();   px->f(); // вызов функции-члена
```

# Оператор [] 2,L,2

**ptr[expr]**
Оператор **[]** доступа к элементу массива.
**ptr** — указатель (имя массива подходит).
**expr** — выражение (целочисленное).
Результат — элемент массива с указанным номером.

```
int m[5]; int i=2;
m[4] = m[i+1] + m[0];

int *p = m+1;
p[2] = 0;

auto ma = new double[50]; // не оператор [], auto = double*
ma[12] = 42.12e4;
delete[] ma;                    // не оператор []

char xs[3][4];
xs[2][3] = 'z';
```

В стандарте: A[B] эквивалентно *(A+B)

**type(list)**
**type{list}** .
**{}** — оператор функционального приведения (functional cast).
**type** — имя типа.
**list** — список аргументов.

```
auto s = Student( "Vasya", 54321, 1 );

auto t = SiegeTank{ "Crusio-20b", 120, 1,2 };
```

**pfun(list)**
**()** — оператор вызова функции.
**pfun** — указатель на функцию.
**list** — список аргументов.
Имя функции без аргументов — это указатель на неё.
Вместо указателя на функцию может быть имя типа — это вызов специальной функции — конструктора.

```
double d = sin(0.5);
double (*f)(double); // указатель на функцию, получающую
                     // и возвращающую double
f = sin;             // указатель на функцию sin

d = f(1.2); // вызов функции по указателю
```

# Операторы ++ – – постфиксные 1,L,2

**lvalue++**
**lvalue– –**
Операторы постфиксного инкремента и декремента.
lvalue — (left value) — реальный объект в памяти.

Результат — **старое** значение аргумента.

Побочное действие (увеличение или уменьшение на 1) выполняется **после** возврата значения (вернуть и изменить).

```
int a = 3, b;
b = a++;
// сейчас a==4, b==3
a = b--;
// сейчас a==3, b==2
```

**typeid(type)**
**typeid(expr)**

Оператор определения типа во время выполнения.
Возвращает что-то, похожее на **const std::type_info&**.

```
#include <typeinfo>
//...
struct A { int x; };
A a;
cout << typeid(A).name() << endl;
```

# Операторы *_cast 2,?,3

Операторы C++ преобразования типа.
Приводят выражение **expr** к типу **type** с учётом ограничений.
**static_cast<type>(expr)** — стандартные безопасные.
**const_cast<type>(expr)** — снятие константности.
**dynamic_cast<type>(expr)** — вниз по дереву наследования с контролем.
**reinterpret_cast<type>(expr)** — зависящие от архитектуры.

```
int a = static_cast<int>(7.2);
int s = 5; const int *ps = &s;
*const_cast<int*>(ps) = 2;
Bird *a = makeBird("Hen"); Hen *h = dynamic_cast<Hen*>(a);
int *p = reinterpret_cast<int*>(0xFF000000);
```

# Операторы ++ – – префиксные 1,R,3

**++lvalue**
**– –lvalue**
Операторы префиксного инкремента и декремента.
lvalue — (left value) — реальный объект в памяти.

Результат — **новое** значение аргумента.

Побочное действие (увеличение или уменьшение на 1) выполняется **до** возврата значения
(изменить и вернуть).

```
int a = 3, b;
b = ++a;
// сейчас a==4, b==4
a = --b;
// сейчас a==3, b==3
```

# Операторы + – унарные 1,R,3

**– expr**
Унарный минус. Результат имеет тот же тип, что и аргумент (должен быть числовой) (rvalue w/o const, volatile) и представляет собой число, противоположное аргументу.

**+expr**
Унарный плюс. Результат имеет тот же тип, что и аргумент и равен ему.

```
int a = 127, b, c;
b = -a; c = +a;
// b == -127; c == 127;
```

# Оператор логического отрицания ! 1,R,3

**! expr**
Оператор **логического** отрицания.

Операнд должен быть логического типа или приводится к нему.
Результат имеет тип **bool**.

```
int a = 4, b = 5;
bool z = ! ( a > b );
// z == true
z = !a;
// z = false

if( ! isalpha(c) ) { /* ... */ };
```

# Оператор побитовой инверсии ~ 1,R,3

**~expr**
Оператор **побитовой** инверсии (дополнения). Операнд должен быть интегрального типа или приводится к нему.
Результат — объект типа **int**, или **unsigned**, или нечто больше. В битовом представлении результата все биты меняют значение на противоположное.

```
int a = 0x55555555, b;
b = ~a;
// a == 0b01010101010101010101010101010101
// b == 0b10101010101010101010101010101010
// 0xAAAAAAAA
int c = ~0;
// c == -1
// 0b11111111111111111111111111111111
```

# Оператор приведения типа (type) C 1,R,3

**(type) expr**

Оператор приведения типа в стиле C.
Смесь из static_cast, const_cast, and reinterpret_cast.
Лучше использовать xxxxxxx_cast.

```
double a = sqrt(14);
int z = (int)a;
int zpp { (int)a };
int *m = (int*) malloc(127); // Good+error C, bad C++
int *x = (int*)z;     // BEWARE!!
```

**&lvalue**

Оператор взятия адреса. Возвращает адрес объекта. Если операнд имеет тип **T**, то результат — тип **T*** (указатель на T).

```
int a = 454;
int *pa = &a;
// int *xxa = &(a+5); // Error: not lvalue

double mv[32];
const double* pd = &(mv[4]); // Ok, lvalue (=mv+4);
```

# Оператор * унарный 1,R,3

***ptr**

Оператор разыменования (доступа по адресу).
Возвращает объект, на который указывает указатель ptr.
Операнд должен быть выражением адресного типа.
Если операнд имеет тип ***T**, то результат — тип **T**.

```
int a = 454; int *pa = &a;
*pa = 712;
// теперь a == 712;

int m[4] = { 7, 5, 4, 3 };
int *t = m;
*(t+2) = 0; // t[2] = 0;
```

# Оператор sizeof 1,R,3

**sizeof(type)**
**sizeof(expr)**
Оператор определения размера объекта, типа или выражения (в байтах).
Тип возвращаемого значения — **const std::size_t**.

```
auto b_i = sizeof(int) * 8;
if( sizeof(short) != sizeof(uint16_t) ) { };
auto lz = sizeof(b_i);
```

Размер массива может определить только в области видимости определения. Нельзя определить размер функции, битового поля и неполного типа.
P.S. C++11: существует **sizeof...**, **alignof**

**new**, **new[]**

Операторы создания объекта и массива объектов в свободной памяти. Возвращает адрес созданных объектов. 2 действия: выделяет память и конструирует объект.

**delete**, **delete[]**

Операторы уничтожения объектов, созданных с помощью new. 2 действия: разрушает объект и освобождает память.

```
int *w = new int;  *w = 3;
int *mw = new int [120]; mw[119] = 54;
Student *s = new Student("Vasya",1234,1);
delete w;
delete[] mw;
delete s; s = nullptr;
```

Созданное надо уничтожать, и только 1 раз! Сам указатель не изменяется. Безопасно применять delete к нулевому указателю (nullptr).

**obj.*ptr_to_member**
**pobj->*ptr_to_member**
Операторы доступа к члену объекта по имени объекта(.*) или указателю на объект (->*) и указателю на член.

```
struct S  {
  int a, b;
  int f(int x) const { cout <<"Fun"<<endl; return x+1;};
};
S sx { 5, 7 };
int S::* pmi = &S::b; // pointer to member
sx.*pmi = 12;
int (S::*pf)(int) const = &S::f;
cout << "sx.b= " << sx.b << endl;
(sx.*pf)( 4 ); // call func. Priority!
```

# Операторы * / % бинарные 2,L,5

**expr * expr**
**expr / expr**
**expr % expr**
Операторы умножения, деления и остатка от целочисленного деления. Тип результата определяется типом аргумента с большими возможностями представления.

```
double a { 1.1 }, b, c;
int i = 4, j, k;
b = a * i; // тип – double
j = i * 'c'; // int
k = i % 3;
```

# Операторы + – бинарные 2,L,6

**expr + expr**
**expr – expr**
Операторы сложения и вычитания. Тип результата определяется типом аргумент с большими возможностями представления.

```
double a { 1.1 }, b, c;
int i = 4, j, k;
b = a + i; // тип – double
j = i – 'c'; // int

double z[7];
double *pz1 = z + 4;
double *pz2 = z + 1;
auto dp = pz2 – pz1; // auto = ptrdiff_t
// pz1 + pz2 : ERROR
```

**expr << expr**
**expr >> expr**

Операторы побитового сдвига. Операнды должны быть целочисленными или пользовательскими. Первый операнд сдвигается влево или вправо на число разрядов, указанных вторым операндом. Биты, вышедшие за пределы разрядной сетки теряются. При сдвиге влево число справа дополняется нулями. При сдвиге вправо знаковое число размножает старший разряд (зависит от реализации), а беззнаковое — дополняется нулями. Результат — не меньше чем **int**. Результат сдвига на отрицательное число, и на число бит, большее, чем размер левого операнда не определён(UB)

```
int a = 5;           // ...00000101
int b = a << 2;      // ...00010100 = 20 = 5*4
int t = -1;          // ...11111111111111
int x = t>>2;        // ...11111111111111
//                        ...^^---- copy from MSB
unsigned u=~0u;      // ...11111111111111
unsigned f = u>>2;   // ...00111111111111
```

# Оператор <=> 2,L,8

**expr <=> expr** — оператор трёхстороннего сравнения
a <=> b возвращает
<0 — если a < b,
>0 — если a > b
==0 — если a равно или эквивалентно b.

```
int x = 5, y = 10;
auto v = x <=> y; // std::strong_ordering
// cout << "v= " << v << endl; // error
if( v < 0 ) {
  cout << "x<y" << endl;
} else if ( v > 0 ) {
  cout << "x>y" << endl;
} else {
  cout << "x==y" << endl;
}
```

Тип возвращаемого значения: **std**::strong_ordering или **std**::partial_ordering.

**expr < expr**
**expr > expr**
**expr <= expr**
**expr >= expr**
Операторы сравнения. Результат имеет тип **bool**.

```
int a = 12, b=4;
bool z = b >= a; // false
if( z ) {
  /* same actions */
}
if( a < 2 ) {
  // more actions
}
```

**expr == expr**
**expr != expr**
Операторы сравнения. Результат имеет тип **bool**.

```
if( a == 12 ) {
 /* ........ */
}

if( sizeof(int) != 4 ) {
  complain();
}
```

# Оператор & бинарный 2,L,11

**expr & expr**

Оператор **побитового** “И” (AND). Операнды — целочисленные. В результате бит установлен в “1”, если **и** в первом, **и** во втором операнде он установлен. (->int).

```
unsigned a = 10, b=0x0B, c; // 0x0B == 12
c = a & b;
// a A 1010
// b C 1100
// c 8 1000
```

Часто используется для того, что бы сбросить (замаскировать) те биты, которые не нужны.

```
unsigned x = ..........................;
unsigned mask = 1<<5;
if( x & mask ) { ................ }
```

Здесь проверяем, установлен ли бит № 6

**expr ^expr**

Оператор побитового исключающего “ИЛИ” (XOR). Операнды — целочисленные. В результате бит установлен в “1”, если **или** в первом, **или** во втором операнде он установлен, **но не в обеих**. (->int).

```
unsigned a = 10, b=0x0B, c; // 0x0B == 12
c = a ^ b;
// a A 1010
// b C 1100
// c 6 0110
```

Часто используется для того, что бы инвертировать заданные биты.

```
unsigned x = ..........................;
unsigned n = ~0u >> 16;
unsigned t = x ^ n;
```

Здесь инвертируются все биты, кроме 16 старших.

**expr |expr**
Оператор побитового “ИЛИ” (OR). Операнды — целочисленные. В результате бит установлен в “1”, если **или** в первом, **или** во втором операнде он установлен, **или в обеих**. (->int).

```
unsigned a = 10, b=0x0B, c; // 0x0B == 12
c = a | b;
// a A 1010
// b C 1100
// c E 1110
```

Часто используется для установки бит.

```
unsigned x = ..........................;
unsigned t = x | 0x0F;
```

Здесь устанавливаются 4 младших бита.

**expr && expr**
Оператор **логического** “И”.
Операнды и результат — логические (**bool**).

```
if( a > 5 && c < 3 ) {
  /*............*/
}
```

Первый операнд вычисляется в первую очередь. Если результат определен первым операндом,то второй не вычисляется.

**expr ||expr**
Оператор **логического** “ИЛИ”.
Операнды и результат — логические (**bool**).

```
if( a > 5 || c < 4) {
  /*............*/
}
```

Первый операнд вычисляется в первую очередь. Если результат определен первым операндом,то второй не вычисляется.

# Оператор ? : 3,R,16

**expr1 ? expr2 : expr3**
Тернарный оператор использования условия.
Если **expr1** — true, то результат — **expr2**, иначе — **expr3**.

```
c = ( a > 0 ) ? a : 0;

cout << ( (x & 1) ? '1' : '0' ) << endl;
```

**expr2** и **expr3** должны иметь совместимые типы.

**lvalue = expr**
**lvalue += expr**, **lvalue –= expr**, **lvalue *= expr**,
**lvalue /= expr**, **lvalue %= expr**, **lvalue <<= expr**,
**lvalue >>= expr**, **lvalue &= expr**, **lvalue ^= expr**,
**lvalue |= expr**

Операторы присваивания и с присваиванием.
Бинарные правоассоциативные операторы с побочным эффектом (присваиванием).

```
a = b = c = 0;
z += 5;
f <<= 4;
a *= v += 4;
```

**throw expr**

Оператор создания исключительной ситуации.

```
if( l >= memSize ) {
  throw out_of_memory;
}
```

**expr1 , expr2**
Оператор последовательности вычисления. Результат **expr1** отбрасывается, результат **expr2** используется. Порядок вычисления гарантируется.

```
for( i=0,j=n-1; i<j; ++i,--j ) {
  a[i] = b[j];
}

f( a, b );    // Not an operator - 2 arguments
f( (a, b) ); // Operator - 1 argument
```

# Замечания к использованию операторов

В некоторых случаях порядок определяется правилами языка:

```
char s[] = "ABCD"; char *p = s;
*p++ = 'X'; // means *p = 'X'; ++p; => "XBCD", *p == 'B'
```

Если в одном выражении применяются операторы с побочным действием к одному объекту — результат не определён!

```
int a = 0;
a += a++ + ++a;        // BAD!
f( a++, a++, a++ );    // BAD!
m[a++]=++a;            // BAD!
```

Для пользовательских типов действия многих операторов можно задать с помощью функций-операторов:

```
string s = "ABCD";
s += "Zqt";
if( s == "XXX" ) {}
```