Operadores

Gonzalo Travieso

2018

1 Preliminares

Agora que vimos como lidar com os dados que compõe o nosso programa, precisamos ver como realizar as operações que desejamos sobre eles.

Toda operação que queremos executar deve ser decomposta numa sequência de operações mais simples, pré-definidas na linguagem. Essas operações básicas são geralmente representadas por **operadores**. Existem diversos tipos de operadores, de acordo com a categoria de operações que desempenham.

Os operadores são representados por símbolos, mas é uma característica de C++ que **um mesmo símbolo pode representar diferentes operações**, de acordo com o tipo de dados ao qual está sendo aplicado. Por exemplo, se encontramos a expressão `a+b`, não sabemos que operação será executada antes de sabermos os tipos de `a` e `b`. Se, por exemplo, `a` e `b` forem `int`, então a operação será a de soma de números em complemento de 2; se `a` e `b` forem `double`, a operação será de soma de números de pontos flutuante de 64 bits como definida pelo padrão IEEE 754.

2 Operadores aritméticos

Para operações aritméticas, temos os operadores `+, -, *, /, %`, correspondendo a soma, subtração, produto, divisão e resto de divisão. C++ não tem um operador para exponenciação (apesar de ter uma função na biblioteca matemática). Note que, no caso de `int`, a divisão é inteira, com aproximação na direção de zero. Esses operadores têm a precedência tradicional da matemática, com os operadores multiplicativos (`*, /, %`) precedendo os operadores aditivos (`+, -`), desde que outra coisa não seja indicada por meio de parêntesis. Operadores de mesma precedência numa expressão são avaliados da esquerda para a direita.

```
int a{3};
int b{4};
double x{3};
double y{4};
auto c = (a/b)*b;
auto z = (x/y)*y;
int t = (a/y)*b;
int w = (x/b)*y;
auto q = a + b * (c - z) / t;
```

Nestas expressões, resulta que `c` é uma variável `int` de valor `0` (pois ao dividirmos `a` por `b` a divisão é inteira, resultando em `0`), enquanto `z` é uma variável `double` de valor `3.0`. Já no caso de `t`, ao realizar a divisão `a/y` o compilador nota que `a` é `int`, enquanto `y` é `double`. Não existe uma divisão que mistura tipos. Então o compilador tem que converter um dos valores para o outro tipo. As regras de conversão são complexas, mas construídas para tentar preservar ao máximo a precisão do valor. Neste caso, é melhor converter o `int` para `double` e realizar a divisão em ponto flutuante; o resultado é depois multiplicado por `b` (que precisa ser convertido para `double`), resultando em `3.0`. Por fim, este valor `double` precisa ser convertido para `int`, que é o tipo especificado para `t`. No cálculo de `w` acontecerá algo similar ao de `t`, mas com uma conversão a menos. No cálculo de `q`, como há misturas entre `int` e `double`, os `int` serão convertidos para `double` e o resultado final será `double`, que será o tipo de `q`; além disso, a ordem de execução das operações será: primeiro a subtração (por causa dos parêntesis), em seguida o produto, depois a divisão e por fim a soma.

3 Operadores de atribuição

Quando queremos colocar um novo valor em uma variável, precisamos usar um operador de atribuição. O mais simples deles é representado pelo operador `=`.

```
c = 2 * a + b;
```

Neste exemplo, colocamos na variável `c` o valor do dobro de `a` somado com `b`.

Além desse, existem diversos operadores de atribuição associada com uma outra operação. Por exemplo, podemos associar atribuição com uma operação aritmética:

```
a += 3*b;
```

Isto corresponde a `a = a + 3 * b`. Da mesma forma com os outros operadores aritméticos

```
b -= 3; // O mesmo que b = b - 3
x /= 5; // O mesmo que x = x / 5;
c %= a; // O mesmo que c = c % a;
z *= a - b + c * d; // O mesmo que z = z * (a - b + c * d)
```

4 Operadores de comparação

Estes são operadores que permitem comparar valores:

`<` menor que

`>` maior que

`<=` menor ou igual a

`>=` maior ou igual a

`==` igual que

`!=` diferente de

Note que esses operadores retornam um `bool` (`true` ou `false`).

```
1 < 2 // true
4 > -5 // true
2 <= 1 // false
2 >= 1 // true
1 == 2 // false
1 != 2 // true
```

5 Operadores lógicos

Operadores lógicos combinam valores `bool` de acordo com os conjuntivos lógicos:

e lógico

|| **ou** lógico

! **negação** (inverte o resultado)

```
auto x{3};
x != 2 && x < 4 // true
x == 2 && x < 4 // false
x == 2 || x < 4 // true
!(x == 2) && x < 4 // true
(2 <= x) && (x <= 4) // true
```

6 Operadores binários (bit a bit)

Estes são similares aos lógicos, mas operam bit a bit na representação binária do valor fornecido:

e bit a bit

| **ou** bit a bit

~ *inversão~ bit a bit

ou exclusivo bit a bit

« deslocamento para a esquerda

» deslocamento para a direita

Os operadores binários podem ser combinados com o = da mesma forma que os operadores aritméticos.

```
auto a{0b110011};
b = a & 0b0101; // b == 0b000001
b = a | 0b0101; // b == 0b110111
b ^= 0b0101; // b == 0b110010
b <<= 2; // b == 0b11001000
b >>= 4; // b == 0b1100
```

7 Auto-incremento e auto-decremento

O C++ também dispõe dos chamados operadores de auto-incremento e auto-decremento, representados por `++` e `--`, respectivamente. Para cada um, existem duas variantes: o pré-incremento e o pós-incremento, diferenciados pela colocação do operador em relação à variável que vai ser incrementada.

```
++a; // pr -incremento; resulta que a = a + 1
--a; // pr -decremento; resulta que a = a - 1
a++; // p s-incremento; resulta que a = a + 1
a--; // p s-decremento; resulta que a = a - 1
```

A diferença entre as versões pré e pós só aparece quando esse operadores são usados em uma expressão. Não estudaremos isso aqui por usaremos a seguinte regra:

Nunca usar auto-incrementos/decrementos em expressões.

Sugiro seguir essa regra pois o uso de auto-incrementos (ou decrementos) em expressões dá origem a dificuldades de compreensão e erros sutis. Quanto às versões pré ou pós, não sendo em expressões elas são quase equivalentes. Recomendo **usar sempre pré-incremento ou decremento**. A razão ficará clara mais tarde.