

Operadores

De atribuição

O operador `=` denota uma atribuição. **Sempre**. Em qualquer contexto - mesmo dentro de um `if` ou `while`. Por exemplo, esse código é válido, mas **não** compara `x` com 3:

```
if(x = 3) {  
    printf("a condicao eh sempre verdadeira! \n");  
}
```

O código é válido porque a expressão `x = 3` tem, na verdade, um valor: `3`, que é o operando da direita. (O operador `=` devolve o operando da direita para permitir atribuições encadeadas, como `a = b = c = 1;` por exemplo.) Esse valor, como qualquer outro valor que não é zero, é considerado como `true`.

Para comparar `x` com 3, seria preciso usar o operador `==`.

Nota: é comum, em C, ver construções do tipo

```
if(((var = function_call(args)) == NULL) {  
    /* tratamento de erro */  
}
```

Esse código é uma versão mais concisa de

```
var = function_call(args);  
if(var == NULL) {  
    /* tratamento de erro */  
}
```

O uso de parênteses ao redor da atribuição (no primeiro exemplo) é devido ao fato de que o operador `=` tem precedência menor que o operador `==`.

Aritméticos

Além das 4 operações básicas (+, -, *, /), há também o operador `%`, que calcula o resto de divisão entre dois inteiros.

Não há nenhum operador para exponenciação: no caso de inteiros pode-se fazer um loop, no caso

de floats, há a função `pow()` (declarada em `math.h`).

Nota: em C, toda divisão entre inteiros também gera um resultado inteiro (a parte decimal é truncada). Por exemplo

```
float x = 17 / 5;
printf("%f\n", x);
```

Imprime 3.000000. Para forçar uma divisão não-inteira, é preciso fazer uma conversão de tipo (cast) de um dos operandos para float:

```
float x = ((float)17) / 5;
printf("%f\n", x);
```

Imprime 3.400000, como esperado.

De comparação

`==` : igual a

`!=` : diferente de

`<` e `>` : menor que e maior que

`<=` e `>=` : menor ou igual e maior ou igual

Nota: em C, esses operadores não podem ser usados com strings. Use a função `strcmp()` em vez disso.

Lógicos

Esses operadores atuam como portas lógicas e permitem fazer condições compostas (novamente, `0` equivale a `false` e não-zero equivale a `true`)

`&&` AND

`||` OR

`!` NOT

Nota: os operadores `&&` e `||` são chamados curto-circuitados, porque podem não avaliar todos os operandos. Por exemplo, em:

```
if( foo() && bar() ) {
    /* alguma coisa */
}
```

Se a chamada `foo()` retornar falso, então a função `bar()` nunca é chamada, pois o resultado final será falso de qualquer forma. Da mesma forma, se tivermos

```
if(foo() || bar()) {  
    /* alguma coisa */  
}
```

E se a chamada `foo()` retornar verdadeiro, então o resultado será necessariamente verdadeiro e a função `bar()` nunca é chamada.

De incremento e decremento

Os operadores `++` e `--` representam incremento e decremento, e aumentam/diminuem o valor da variável em 1, respectivamente.

Os dois operadores têm uma peculiaridade: ambos podem aparecer antes ou depois da variável (`++var` e `var++`, `--var` e `var--`). Por exemplo:

```
int x = 4;  
int y = ++x; /* pré-incremento: realizado antes da atribuição */  
printf("%d\n", y);
```

Imprime 5, enquanto

```
int x = 4;  
int y = x++; /* pós-incremento: realizado depois da atribuição */  
printf("%d\n", y);
```

Imprime 4. O pré-decremento e pós-decremento funcionam de forma análoga.

Esses operadores são usados ostensivamente em C - em índices de loops `for`, em ponteiros, em índices de arrays, etc.

De atribuição composta

Todos os operadores aritméticos e bitwise possuem uma versão de atribuição composta, que são da forma `operador=`. Por exemplo:

```
x += 5;
```

Que é, por definição, equivalente a

```
x = x + 5;
```

Os operadores de atribuição composta são:

`+=` `-=` `*=` `/=` `%=` `&=` `|=` `^=` `<<=` `>>=`

sizeof

O operador `sizeof` devolve o tamanho do operando, em bytes. Exemplos:

```
/* size_t é o tipo do resultado de sizeof */
size_t int_size = sizeof(int); /* sizeof(tipo) */
int x;
size_t x_size = sizeof(x); /* sizeof(variável) */
size_t expr_size = sizeof(&x); /* sizeof(expressão) */
```

Por definição, `sizeof(char)` vale 1; portanto, `char` é equivalente a byte.

Esse operador pode ser usado para descobrir quantos elementos uma array tem:

```
int arr[20];
/* ... */

/* sizeof(arr) == 20 * sizeof(int) e sizeof(*arr) == sizeof(int) */
size_t arr_size = sizeof(arr) / sizeof(*arr);
```

(Note, porém, que essa técnica não funciona com memória alocada dinamicamente, nem com arrays que foram passadas para uma função - funções não recebem arrays, mas sim ponteiros.)

Bitwise

Operadores bitwise atuam nos bits individuais de seus operandos.

O operador `~` (NOT) tem apenas um operando e inverte todos os seus bits.

Os operadores `&` (AND), `|` (OR) e `^` (XOR) têm dois operandos, e atuam como portas lógicas em seus bits: no caso do `&`, o primeiro bit do resultado é o AND do primeiro bit dos dois operandos, e assim sucessivamente para os demais bits. Por exemplo:

```
int x = 0xb; /* 1011 em binario */
int y = 0x5; /* 0101 em binario */
int z = x & y;
printf("%d\n", z);
```

Imprime 1:

b3	b2	b1	b0	
1	0	1	1	x
0	1	0	1	y
				&
0	0	0	1	z

O operador `|` é frequentemente usado para obter combinações de uma ou mais *flags*, que podem estar ativas ou não, e por isso podem ser representadas por um único bit. O uso de *flags* (em vez de uma array de `bool`s, por exemplo), permite fazer combinações de inúmeras de *flags* usando apenas uma variável.

Por exemplo, no caso da função `open()` (de UNIX):

```
int fd = open("log.txt", O_WRONLY|O_APPEND|O_CREAT);
/* ... */
close(fd);
```

Cada macro `O_*` usada como operando do `|` é uma *flag*, e corresponde a um valor que tem todos os bits zero, exceto por um bit em uma posição específica (e essa posição é diferente para cada macro). Então fazer `|` com essa macro ativa o bit naquela posição.

De maneira similar, podemos usar o operador `&` para checar se uma flag está ativa:

```
if(flags & O_CREAT) {
    /* flag O_CREAT está ativa, aja de acordo */
}
```

Note que a expressão usada como condição no `if` será não-zero (`true`) se e somente se o bit correspondente em `flags` está ativo.

Além desses operadores bitwise, há os de bitshift: `<<` e `>>`, que denotam *shift* para a esquerda e para a direita, respectivamente. A operação de bitshift desloca os bits do primeiro operando, em um número inteiro de posições, denotado pelo segundo operando. Por exemplo:

```
int x = 7 << 2;
printf("%d\n", x);
```

Imprime 28:

b31	b30	...	b5	b4	b3	b2	b1	b0	
0	0	0	0	0	0	1	1	1	7
0	0	0	0	1	1	1			7 << 2

As posições vazias que são criadas à esquerda (b1 e b0) são preenchidas com zeros, e os bits mais à esquerda (b31 e b30, em **negrito**) são descartados. (Supomos, nesse exemplo e nos seguintes, que um `int` tem 32 bits.)

No *shift* para a esquerda, os bits vazios são sempre preenchidos com zero, mesmo quando os bits mais significativos valem 1:

```
int x = -3 << 3;
printf("%d\n", x);
```

Imprime -24:

b31	b30	...	b5	b4	b3	b2	b1	b0	
1	1	1	1	1	1	1	0	1	-3
1	1	1	1	0	1	0	0	0	-3 << 3

Portanto, um *shift* para a esquerda em N posições equivale a multiplicar um inteiro por 2^N , independentemente de seu sinal.

Consideremos agora um *shift* para a direita, por exemplo, `15 >> 2`:

b31	b30	...	b5	b4	b3	b2	b1	b0	
0	0	0	0	0	1	1	1	1	15
		0	0	0	0	0	1	1	15 >> 2

Nesse caso, criam-se posições vagas nos bits b31 e b30, e os bits b1 e b0 são desprezados.

Preenchendo os bits vazios com zero, temos como resultado o valor 3, que é igual a $\text{floor}(15 / 2^2)$.

Note que, no caso do *shift* à direita, nem sempre podemos preencher os bits vazios com zero.

Considere, por exemplo, `-3 >> 1`:

[illegible]

b31	b30	...	b5	b4	b3	b2	b1	b0	
	1	1	1	1	1	1	1	0	-3 >> 1

Se preencheremos a posição vazia com zero, teremos alterado o sinal do resultado! Por outro lado, esses mesmos bits podem corresponder a um `unsigned int`: nesse caso o bit b31 não é um bit de sinal, e devemos preencher as posições vazias com bits zero. Daí, surge uma diferenciação entre *shift* à direita lógico e aritmético:

- no *shift* à direita lógico, as posições vazias são sempre preenchidas com zeros.
- no *shift* à direita aritmético, as posições vazias são preenchidas com o bit mais à esquerda, que é interpretado como bit de sinal.

Em C, o operador `>>` denota *shift* à direita lógico quando o operando é de tipo `unsigned`. Caso contrário, o padrão não especifica se o tipo de *shift* usado deve ser lógico ou aritmético.

Portanto, `x >> N` equivale a `floor(x / 2^N)`, contanto que o tipo adequado de *shift* seja usado.

Precedência e associatividade

A precedência dos operadores dita qual operação é realizada primeiro em uma expressão. Considere, por exemplo:

```
int x = 4 + 5 * 2;
```

Intuitivamente, a multiplicação deve acontecer antes da soma. Para que isso aconteça, o operador `*` tem precedência maior que o `+`, de modo que a expressão equivale a

```
int x = 4 + (5 * 2);
```

Se quisermos que a soma aconteça antes, é preciso colocá-la entre parênteses:

```
int x = (4 + 5) * 2;
```

A precedência dos operadores aritméticos é intuitiva, mas isso não necessariamente ocorre para outros tipos de operadores; por isso, é preciso conhecer sua precedência.

A associatividade também dita a ordem das operações - mas quando um mesmo operador aparece várias vezes. Considere, por exemplo:

```
int x = 1 + 2 + 3;
```

Essa expressão será avaliada como `(1 + 2) + 3` ou como `1 + (2 + 3)`? No primeiro caso, temos associatividade da direita para a esquerda, pois avaliamos a expressão nesse sentido; no segundo,

a associatividade é da esquerda para a direita. Nesse exemplo, a associatividade não faz diferença, pois o resultado será o mesmo nos dois casos. Mas considere agora:

```
int a = 0, b = 0, c = 0;
a = b = c = 1;
```

Se tentarmos avaliar a expressão da esquerda para a direita, teremos um problema: a atribuição `a = b` dará à variável `a` o valor de `b`, que ainda é `0`. Porém, se avaliarmos a expressão no sentido contrário, tudo funciona: `a = (b = (c = 1));` -> `a = (b = 1);` -> `a = 1;` -> `1`. (Note que o valor final dessa expressão é descartado; o que importa para nós são as atribuições feitas ao longo do caminho.) Portanto, o operador `=` precisa ter precedência da direita para a esquerda.

Na tabela abaixo, operadores na mesma linha têm a mesma precedência, e operadores em linhas mais acima têm precedência maior.

TABLE 2-1. PRECEDENCE AND ASSOCIATIVITY OF OPERATORS

OPERATORS	ASSOCIATIVITY
<code>() [] -> .</code>	left to right
<code>! ~ ++ -- + - * & (type) sizeof</code>	right to left
<code>* / %</code>	left to right
<code>+ -</code>	left to right
<code><< >></code>	left to right
<code>< <= > >=</code>	left to right
<code>== !=</code>	left to right
<code>&</code>	left to right
<code>^</code>	left to right
<code> </code>	left to right
<code>&&</code>	left to right
<code> </code>	left to right
<code>? :</code>	right to left
<code>= += -= *= /= %= &= ^= = <<= >>=</code>	right to left
<code>,</code>	left to right

Unary +, -, and * have higher precedence than the binary forms.

Fonte: Brian W. Kernighan, Dennis M. Ritchie. The C Programming Language. 2ª edição.

(Nota: os operadores `?` e `:` são mencionados na seção Condicionais; o operador `[]` é mencionado na subseção Arrays; os operadores `&` e `*` (unário) são mencionados na seção

Ponteiros; os operadores `.` e `->` serão mencionados na seção Tipos definidos pelo usuário.)

Revision #1

Created Mon, Jan 28, 2019 11:32 PM by Luana

Updated Mon, Jan 28, 2019 11:33 PM by Luana