2. Expressões

W. Celes e J. L. Rangel

Em C, uma expressão é uma combinação de variáveis, constantes e operadores que pode ser avaliada computacionalmente, resultando em um valor. O valor resultante é chamado de *valor da expressão*.

2.1. Variáveis

Podemos dizer que uma variável representa um espaço na memória do computador para armazenar determinado tipo de dado. Na linguagem C, todas as variáveis devem ser explicitamente declaradas. Na declaração de uma variável, obrigatoriamente, devem ser especificados seu *tipo* e seu *nome*: o nome da variável serve de referência ao dado armazenado no espaço de memória da variável e o tipo da variável determina a natureza do dado que será armazenado.

Tipos básicos

A linguagem C oferece alguns tipos básicos. Para armazenar valores inteiros, existem três tipos básicos: `char`, `short int`, `long int`. Estes tipos diferem entre si pelo espaço de memória que ocupam e conseqüentemente pelo intervalo de valores que podem representar. O tipo `char`, por exemplo, ocupa 1 byte de memória (8 bits), podendo representar 2^8 (=256) valores distintos. Todos estes tipos podem ainda ser modificados para representarem apenas valores positivos, o que é feito precedendo o tipo com o modificador "sem sinal" – `unsigned`. A tabela abaixo compara os tipos para valores inteiros e suas representatividades.

Tipo	Tamanho	Representatividade
char	1 byte	-128 a 127
unsigned char	1 byte	0 a 255
short int	2 bytes	-32 768 a 32 767
unsigned short int	2 bytes	0 a 65 535
long int	4 bytes	-2 147 483 648 a 2 147 483 647
unsigned long int	4 bytes	4 294 967295

Os tipos `short int` e `long int` podem ser referenciados simplesmente com `short` e `long`, respectivamente. O tipo `int` puro é mapeado para o tipo inteiro natural da máquina, que pode ser `short` ou `long`. A maioria das máquinas que usamos hoje funcionam com processadores de 32 bits e o tipo `int` é mapeado para o inteiro de 4 bytes (`long`).[1]

O tipo `char` é geralmente usado apenas para representar códigos de caracteres, como veremos nos capítulos subseqüentes.

[1] Um contra-exemplo é o compilador TurboC, que foi desenvolvido para o sistema operacional DOS mas ainda pode ser utilizado no Windows. No TurboC, o tipo `int` é mapeado para 2 bytes.

A linguagem oferece ainda dois tipos básicos para a representação de números reais (ponto flutuante): `float` e `double`. A tabela abaixo compara estes dois tipos.

Tipo	Tamanho	Representatividade
float	4 bytes	$\pm 10^{-38}$ a 10^{38}
double	8 bytes	$\pm 10^{-308}$ a 10^{308}

Declaração de variáveis

Para armazenarmos um dado (valor) na memória do computador, devemos reservar o espaço correspondente ao tipo do dado a ser armazenado. A declaração de uma variável reserva um espaço na memória para armazenar um dado do tipo da variável e associa o nome da variável a este espaço de memória.

```
int a;                  /* declara uma variável do tipo int */
int b;                  /* declara outra variável do tipo int */
float c;                /* declara uma variável do tipo float */

a = 5;                  /* armazena o valor 5 em a */
b = 10;                 /* armazena o valor 10 em b */
c = 5.3;                /* armazena o valor 5.3 em c */
```

A linguagem permite que variáveis de mesmo tipo sejam declaradas juntas. Assim, as duas primeiras declarações acima poderiam ser substituídas por:

```
int a, b;               /* declara duas variáveis do tipo int */
```

Uma vez declarada a variável, podemos armazenar valores nos respectivos espaços de memória. Estes valores devem ter o mesmo tipo da variável, conforme ilustrado acima. Não é possível, por exemplo, armazenar um número real numa variável do tipo `int`. Se fizermos:

```
int a;
a = 4.3;                /* a variável armazenará o valor 4 */
```

será armazenada em `a` apenas a parte inteira do número real, isto é, `4`. Alguns compiladores exibem uma advertência quando encontram este tipo de atribuição.

Em C, as variáveis podem ser inicializadas na declaração. Podemos, por exemplo, escrever:

```
int a = 5, b = 10;      /* declara e inicializa as variáveis */
float c = 5.3;
```

Valores constantes

Em nossos códigos, usamos também valores constantes. Quando escrevemos a atribuição:

```
a = b + 123;
```

sendo `a` e `b` variáveis supostamente já declaradas, reserva-se um espaço para armazenar a constante 123. No caso, a constante é do tipo inteiro, então um espaço de quatro bytes (em geral) é reservado e o valor 123 é armazenado nele. A diferença básica em relação às variáveis, como os nomes dizem (variáveis e constantes), é que o valor armazenado numa área de constante não pode ser alterado.

As constantes também podem ser do tipo real. Uma constante real deve ser escrita com um ponto decimal ou valor de expoente. Sem nenhum sufixo, uma constante real é do tipo `double`. Se quisermos uma constante real do tipo `float`, devemos, a rigor, acrescentar o sufixo `F` ou `f`. Alguns exemplos de constantes reais são:

```
12.45      constante real do tipo double
1245e-2    constante real do tipo double
12.45F     constante real do tipo float
```

Alguns compiladores exibem uma advertência quando encontram o código abaixo:

```
float x;
...
x = 12.45;
```

pois o código, a rigor, armazena um valor `double` (`12.45`) numa variável do tipo `float`. Desde que a constante seja representável dentro de um `float`, não precisamos nos preocupar com este tipo de advertência.

Variáveis com valores indefinidos

Um dos erros comuns em programas de computador é o uso de variáveis cujos valores ainda estão indefinidos. Por exemplo, o trecho de código abaixo está errado, pois o valor armazenado na variável `b` está indefinido e tentamos usá-lo na atribuição a `c`. É comum dizermos que b tem "lixo".

```
int a, b, c;
a = 2;
c = a + b;            /* ERRO: b tem "lixo" */
```

Alguns destes erros são óbvios (como o ilustrado acima) e o compilador é capaz de nos reportar uma advertência; no entanto, muitas vezes o uso de uma variável não definida fica difícil de ser identificado no código. Repetimos que é um erro comum em programas e uma razão para alguns programas funcionarem na parte da manhã e não funcionarem na parte da tarde (ou funcionarem durante o desenvolvimento e não funcionarem quando entregamos para nosso cliente!). Todos os erros em computação têm lógica. A razão de o programa poder funcionar uma vez e não funcionar outra é que, apesar de indefinido, o valor da variável existe. No nosso caso acima, pode acontecer que o valor armazenado na memória ocupada por `b` seja `0`, fazendo com que o programa funcione. Por outro lado, pode acontecer de o valor ser `-293423` e o programa não funcionar.

2.2. Operadores

A linguagem C oferece uma gama variada de operadores, entre binários e unários. Os operadores básicos são apresentados a seguir.

Operadores Aritméticos

Os operadores aritméticos binários são: `+`, `-`, `*`, `/` e o operador módulo `%`. Há ainda o operador unário `-`. A operação é feita na precisão dos operandos. Assim, a expressão `5/2` resulta no valor `2`, pois a operação de divisão é feita em precisão inteira, já que os dois operandos (`5` e `2`) são constantes inteiras. A divisão de inteiros trunca a parte fracionária, pois o valor resultante é sempre do mesmo tipo da expressão. Conseqüentemente, a expressão `5.0/2.0` resulta no valor real `2.5` pois a operação é feita na precisão real (`double`, no caso).

O operador módulo, `%`, não se aplica a valores reais, seus operandos devem ser do tipo inteiro. Este operador produz o resto da divisão do primeiro pelo segundo operando. Como exemplo de aplicação deste operador, podemos citar o caso em que desejamos saber se o valor armazenado numa determinada variável inteira `x` é par ou ímpar. Para tanto, basta analisar o resultado da aplicação do operador `%`, aplicado à variável e ao valor dois.

```
x % 2          se resultado for zero      ⇒ número é par
x % 2          se resultado for um        ⇒ número é ímpar
```

Os operadores `*`, `/` e `%` têm precedência maior que os operadores `+` e `-`. O operador `-` unário tem precedência maior que `*`, `/` e `%`. Operadores com mesma precedência são avaliados da esquerda para a direita. Assim, na expressão:

```
a + b * c /d
```

executa-se primeiro a multiplicação, seguida da divisão, seguida da soma. Podemos utilizar parênteses para alterar a ordem de avaliação de uma expressão. Assim, se quisermos avaliar a soma primeiro, podemos escrever:

```
(a + b) * c /d
```

Uma tabela de precedência dos operadores da linguagem C é apresentada no final desta seção.

Operadores de atribuição

Na linguagem C, uma atribuição é uma expressão cujo valor resultante corresponde ao valor atribuído. Assim, da mesma forma que a expressão:

```
5 + 3
```

resulta no valor 8, a atribuição:

```
a = 5
```

resulta no valor 5 (além, é claro, de armazenar o valor 5 na variável `a`). Este tratamento das atribuições nos permite escrever comandos do tipo:

```
y = x = 5;
```

Neste caso, a ordem de avaliação é da direita para a esquerda. Assim, o computador avalia `x = 5`, armazenando `5` em `x`, e em seguida armazena em `y` o valor produzido por `x = 5`, que é `5`. Portanto, ambos, `x` e `y`, recebem o valor `5`.

A linguagem também permite utilizar os chamados operadores de atribuição compostos. Comandos do tipo:

```
i = i + 2;
```

em que a variável à esquerda do sinal de atribuição também aparece à direita, podem ser escritas de forma mais compacta:

```
i += 2;
```

usando o operador de atribuição composto `+=`. Analogamente, existem, entre outros, os operadores de atribuição: `-=`, `*=`, `/=`, `%=`. De forma geral, comandos do tipo:

```
var op= expr;
```

são equivalentes a:

```
var = var op (expr);
```

Salientamos a presença dos parênteses em torno de `expr`. Assim:

```
x *= y + 1;
```

equivale a

```
x = x * (y + 1)
```

e não a

```
x = x * y + 1;
```

Operadores de incremento e decremento

A linguagem C apresenta ainda dois operadores não convencionais. São os operadores de incremento e decremento, que possuem precedência comparada ao `-` unário e servem para incrementar e decrementar uma unidade nos valores armazenados nas variáveis. Assim, se `n` é uma variável que armazena um valor, o comando:

```
n++;
```

incrementa de uma unidade o valor de `n` (análogo para o decremento em `n--`). O aspecto não usual é que `++` e `--` podem ser usados tanto como operadores pré-fixados (antes da variável, como em `++n`) ou pós-fixados (após a variável, como em `n++`). Em ambos os casos, a variável `n` é incrementada. Porém, a expressão `++n` incrementa `n` *antes* de usar seu valor, enquanto `n++` incrementa `n` *após* seu valor ser usado. Isto significa que, num contexto onde o valor de `n` é usado, `++n` e `n++` são diferentes. Se `n` armazena o valor `5`, então:

```
x = n++;
```

atribui `5` a `x`, mas:

```
x = ++n;
```

atribuiria `6` a `x`. Em ambos os casos, `n` passa a valer `6`, pois seu valor foi incrementado em uma unidade. Os operadores de incremento e decremento podem ser aplicados somente em variáveis; uma expressão do tipo `x = (i + 1)++` é ilegal.

A linguagem C oferece diversas formas compactas para escrevermos um determinado comando. Neste curso, procuraremos evitar as formas compactas pois elas tendem a dificultar a compreensão do código. Mesmo para programadores experientes, o uso das formas compactas deve ser feito com critério. Por exemplo, os comandos:

```
a = a + 1;
a += 1;
a++;
++a;
```

são todos equivalentes e o programador deve escolher o que achar mais adequado e simples. Em termos de desempenho, qualquer compilador razoável é capaz de otimizar todos estes comandos da mesma forma.

Operadores relacionais e lógicos

Os operadores relacionais em C são:

`<`	*menor que*
`>`	*maior que*
`<=`	*menor ou igual que*
`>=`	*maior ou igual que*
`==`	*igual a*
`!=`	*diferente de*

Estes operadores comparam dois valores. O resultado produzido por um operador relacional é zero ou um. Em C, não existe o tipo booleano (*true* ou *false*). O valor zero é interpretado como falso e qualquer valor diferente de zero é considerado verdadeiro. Assim, se o

resultado de uma comparação for falso, produz-se o valor 0, caso contrário, produz-se o valor 1.

Os operadores lógicos combinam expressões booleanas. A linguagem oferece os seguintes operadores lógicos:

`&&`	*operador binário E (AND)*
`||`	*operador binário OU (OR)*
!	*operador unário de NEGAÇÃO (NOT)*

Expressões conectadas por `&&` ou `||` são avaliadas da esquerda para a direita, e a avaliação pára assim que a veracidade ou falsidade do resultado for conhecida. Recomendamos o uso de parênteses em expressões que combinam operadores lógicos e relacionais.

Os operadores relacionais e lógicos são normalmente utilizados para tomada de decisões. No entanto, podemos utilizá-los para atribuir valores a variáveis. Por exemplo, o trecho de código abaixo é válido e armazena o valor `1` em `a` e `0` em `b`.

```
int a, b;
int c = 23;
int d = c + 4;

a = (c < 20) || (d > c);   /* verdadeiro */
b = (c < 20) && (d > c);   /* falso */
```

Devemos salientar que, na avaliação da expressão atribuída à variável `b`, a operação `(d>c)` não chega a ser avaliada, pois independente do seu resultado a expressão como um todo terá como resultado `0` (falso), uma vez que a operação `(c<20)` tem valor falso.

Operador `sizeof`

Outro operador fornecido por C, `sizeof`, resulta no número de bytes de um determinado tipo. Por exemplo:

```
int a = sizeof(float);
```

armazena o valor `4` na variável `a`, pois um `float` ocupa 4 bytes de memória. Este operador pode também ser aplicado a uma variável, retornando o número de bytes do tipo associado à variável.

Conversão de tipo

Em C, como na maioria das linguagens, existem conversões automáticas de valores na avaliação de uma expressão. Assim, na expressão `3/1.5`, o valor da constante `3` (tipo `int`) é promovido (convertido) para `double` antes de a expressão ser avaliada, pois o segundo operando é do tipo `double` (`1.5`) e a operação é feita na precisão do tipo mais representativo.

Quando, numa atribuição, o tipo do valor atribuído é diferente do tipo da variável, também há uma conversão automática de tipo. Por exemplo, se escrevermos:

```
int a = 3.5;
```

o valor `3.5` é convertido para inteiro (isto é, passa a valer `3`) antes de a atribuição ser efetuada. Como resultado, como era de se esperar, o valor atribuído à variável é `3` (inteiro). Alguns compiladores exibem advertências quando a conversão de tipo pode significar uma perda de precisão (é o caso da conversão real para inteiro, por exemplo).

O programador pode explicitamente requisitar uma conversão de tipo através do uso do operador de molde de tipo (operador *cast*). Por exemplo, são válidos (e isentos de qualquer advertência por parte dos compiladores) os comandos abaixo.

```
int a, b;
a = (int) 3.5;
b = (int) 3.5 % 2;
```

Precedência e ordem de avaliação dos operadores

A tabela abaixo mostra a precedência, em ordem decrescente, dos principais operadores da linguagem C.

Operador	Associatividade
( ) [ ] -> .	esquerda para direita
! ~ ++ -- - (tipo) * & sizeof(tipo)	direita para esquerda
* / %	esquerda para direita
+ -	esquerda para direita
<< >>	esquerda para direita
< <= > >=	esquerda para direita
== !=	esquerda para direita
&	esquerda para direita
^	esquerda para direita
|	esquerda para direita
&&	esquerda para direita
||	esquerda para direita
?:	direita para esquerda
= += -= etc.	direita para esquerda
,	esquerda para direita

2.3. Entrada e saída básicas

A linguagem C não possui comandos de entrada e saída do tipo READ e WRITE encontrados na linguagem FORTRAN. Tudo em C é feito através de funções, inclusive as operações de entrada e saída. Por isso, já existe em C uma biblioteca padrão que possui as funções básicas normalmente necessárias. Na biblioteca padrão de C, podemos, por exemplo, encontrar funções matemáticas do tipo raiz quadrada, seno, cosseno, etc., funções para a manipulação de cadeias de caracteres e funções de entrada e saída. Nesta seção, serão apresentadas as duas funções básicas de entrada e saída disponibilizadas pela biblioteca padrão. Para utilizá-las, é necessário incluir o *protótipo* destas funções no

código. Este assunto será tratado em detalhes na seção sobre funções. Por ora, basta saber que é preciso escrever:

```
#include <stdio.h>
```

no início do programa que utiliza as funções da biblioteca de entrada e saída.

Função `printf`

A função `printf` possibilita a saída de valores (sejam eles constantes, variáveis ou resultado de expressões) segundo um determinado formato. Informalmente, podemos dizer que a forma da função é:

`printf (`*formato*`,` *lista de constantes/variáveis/expressões...*`);`

O primeiro parâmetro é uma cadeia de caracteres, em geral delimitada com aspas, que especifica o formato de saída das constantes, variáveis e expressões listadas em seguida. Para cada valor que se deseja imprimir, deve existir um especificador de formato correspondente na cadeia de caracteres *`formato`*. Os especificadores de formato variam com o tipo do valor e a precisão em que queremos que eles sejam impressos. Estes especificadores são precedidos pelo caractere `%` e podem ser, entre outros:

`%c`	*especifica um char*
`%d`	*especifica um int*
`%u`	*especifica um unsigned int*
`%f`	*especifica um double (ou float)*
`%e`	*especifica um double (ou float) no formato científico*
`%g`	*especifica um double (ou float) no formato mais apropriado (%f ou %e)*
`%s`	*especifica uma cadeia de caracteres*

Alguns exemplos:

```
printf ("%d %g\n", 33, 5.3);
```

tem como resultado a impressão da linha:

```
33 5.3
```

Ou:

```
printf ("Inteiro = %d   Real = %g\n", 33, 5.3);
```

com saída:

```
Inteiro = 33   Real = 5.3
```

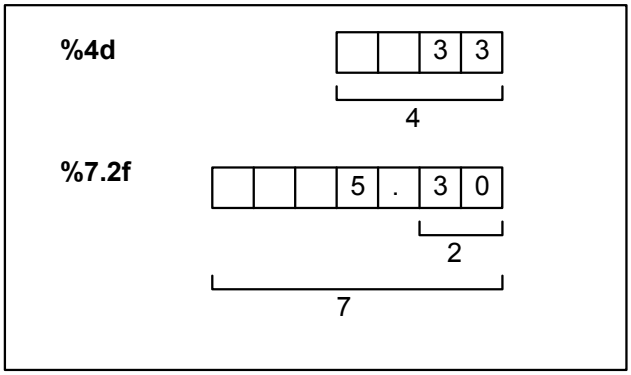
Isto é, além dos especificadores de formato, podemos incluir textos no formato, que são mapeados diretamente para a saída. Assim, a saída é formada pela cadeia de caracteres do formato onde os especificadores são substituídos pelos valores correspondentes.

Existem alguns caracteres de escape que são freqüentemente utilizados nos formatos de saída. São eles:

`\n`	*caractere de nova linha*
`\t`	*caractere de tabulação*
`\r`	*caractere de retrocesso*
`\"`	*o caractere* "
`\\`	*o caractere* \

Ainda, se desejarmos ter como saída um caractere `%`, devemos, dentro do formato, escrever `%%`.

É possível também especificarmos o tamanho dos campos:



A função `printf` retorna o número de campos impressos. Salientamos que para cada constante, variável ou expressão listada devemos ter um especificador de formato apropriado.

Função `scanf`

A função `scanf` permite capturarmos valores fornecidos via teclado pelo usuário do programa. Informalmente, podemos dizer que sua forma geral é:

`scanf` (*formato*`,` *lista de endereços das variáveis...*)`;`

O formato deve possuir especificadores de tipos similares aos mostrados para a função `printf`. Para a função `scanf`, no entanto, existem especificadores diferentes para o tipo `float` e o tipo `double`:

`%c`	*especifica um char*
`%d`	*especifica um int*
`%u`	*especifica um unsigned int*
`%f,%e,%g`	*especificam um float*
`%lf, %le, %lg`	*especificam um double*
`%s`	*especifica uma cadeia de caracteres*

A principal diferença é que o formato deve ser seguido por uma lista de endereços de variáveis (na função `printf` passamos os valores de constantes, variáveis e expressões). Na seção sobre ponteiros, este assunto será tratado em detalhes. Por ora, basta saber que, para ler um valor e atribuí-lo a uma variável, devemos passar o endereço da variável para a função `scanf`. O operador `&` retorna o endereço de uma variável. Assim, para ler um inteiro, devemos ter:

```
int n;
scanf ("%d", &n);
```

Para a função `scanf`, os especificadores `%f`, `%e` e `%g` são equivalentes. Aqui, caracteres diferentes dos especificadores no formato servem para cercar a entrada. Por exemplo:

```
scanf ("%d:%d", &h, &m);
```

obriga que os valores (inteiros) fornecidos sejam separados pelo caractere dois pontos (`:`). Um espaço em branco dentro do formato faz com que sejam "pulados" eventuais brancos da entrada. Os especificadores `%d`, `%f`, `%e` e `%g` automaticamente pulam os brancos que precederem os valores numéricos a serem capturados. A função `scanf` retorna o número de campos lidos com sucesso.