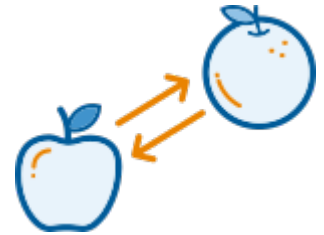


Conversão de tipos

Video desta aula

Muitas vezes, expressões lógicas/aritméticas envolvem valores de tipos distintos, como char, int, float e double, que devem ser convertidos durante o cálculo.



Algumas conversões são efetuadas implicitamente pelo compilador, sem riscos para a integridade dos dados manipulados. Todavia, certas conversões podem levar a resultados errados ou à perda de informação e precisam ser “forçadas” pelo programador, através de conversões explícitas.

Conversão implícita

Quando uma operação lógica/aritmética envolve dois operandos de tipos distintos, o compilador primeiro os converte para um único tipo, antes de avaliar a operação. Nos caso de tipos numéricos, essa conversão é denominada **promoção automática**,

Na promoção automática, o tipo de menor tamanho (com menos bytes) é automaticamente convertido (“promovido”) para o tipo de maior tamanho (com mais bytes).

A hierarquia de tipos considerada para a promoção automática é a seguinte (os tipos unsigned estão no mesmo nível de hierarquia que seus equivalentes signed):

```
char < short < int < long < long long < float < double < long double
```



A regra de promoção automática é aplicada separadamente **A CADA OPERAÇÃO** da expressão lógica/aritmética.

O código a seguir mostra alguns exemplos de promoção automática:

```
char    c = 'A' ;
int     i = 3   ;
short   s = 31  ;
float   f = 31.8 ;
double  d ;

i = c + 30 ;    // o valor de c (65) é convertido de char para int

d = i * f + s ; // ao avaliar i*f,      i é promovido de int a float
                // ao avaliar float + s, s é promovido de short a float
```

A conversão implícita (ou automática) é realizada para as operações aritméticas (+ - * / %), relacionais (< <= > >= == !=) e com bits (& | ^).

Os operadores de atribuição (=, +=, ...) também recebem uma conversão implícita de tipo, para compatibilizar o valor calculado (lado direito da atribuição) com a variável que irá recebê-lo (lado esquerdo).

Erros de conversão

A regra de promoção automática é aplicada a cada operação da expressão lógica/aritmética. Por isso, deve ser tomado cuidado em expressões mais complexas. O código abaixo traz um exemplo dos riscos envolvidos:

[erro-conversao.c](#)

```
#include <stdio.h>

int main ()
{
    int i    = 100 ;
    int j    = 6  ;
    float x  = 6  ;
    float y  ;

    y = i / j * x ;           // (int / int) * float -> int * float
    printf ("y vale %f\n", y) ; // ^^^^^^^^ divisão inteira!

    y = x * i / j ;           // (float * int) / int
    printf ("y vale %f\n", y) ; // resultado correto (vide abaixo)
}
```

Ao avaliar primeiro a operação i/j , ocorre um erro de arredondamento:

```
$ ./a.out
y vale 96.000000
y vale 100.000000
```

As expressões são usualmente avaliadas da esquerda para a direita, mas isso depende da precedência dos operadores, além do compilador e do nível de otimização usado. Por isso, a ordem de avaliação das operações em uma expressão envolvendo tipos distintos deve ser explicitada usando **parênteses**:

```
y = (x * i) / j ;           // (float * int) / int
printf ("y vale %f\n", y) ; // resultado correto sempre
```

A conversão implícita na atribuição também pode provocar perda de informação. O exemplo abaixo mostra algumas dessas situações:

[perdas.c](#)

```
#include <stdio.h>

int main ()
{
    char    c ;
    int     i ;
    long    l ;
    float   f ;
    double  d ;

    c = 'A' ;
    i = c ;    // ok, pois int > char
```

```
printf ("c = %d, i = %d\n", c, i) ;

i = 34 ;
f = i ;      // ok, pois float > int
printf ("i = %d, f = %f\n", i, f) ;

l = 214748364347243 ;
f = l ;      // perda de precisão
printf ("l = %ld, f = %f\n", l, f) ;

l = 214748364347243 ;
d = l ;      // ok, precisão suficiente
printf ("l = %ld, d = %lf\n", l, d) ;

f = 451.28 ;
i = f ;      // parte fracionária é truncada
printf ("f = %f, i = %d\n", f, i) ;

d = 3.141592653589793264 ;
f = d ;      // perda de precisão
printf ("d = %.15f, f = %.15f\n", d, f) ;

l = 12345677890 ;
i = l ;      // perda dos bits mais significativos
printf ("l = %ld, i = %d\n", l, i) ;
}
```



Observe que a precisão de um float é menor que a de um int a partir de $2^{23}+1$, pois a mantissa do float tem apenas 23 bits (padrão IEEE 754). Assim, no caso de um valor inteiro muito grande ($> 2^{23}$), a conversão do mesmo em float implica em perda de precisão.

Conversão explícita

Em alguns casos, é necessário forçar a conversão de tipos de dados, para que uma expressão seja avaliada da forma correta.

Por exemplo:

[media-erro.c](#)

```
#include <stdio.h>

int main ()
{
    int soma, num ;
    float media ;

    soma = 10 ;
    num = 4 ;

    media = soma / num ;
}
```

```
printf ("media = %f\n", media) ;  
}
```

No caso acima, a expressão `soma / num` será avaliada como `int / int`, resultando em uma divisão inteira e consequente perda de precisão. Para gerar o resultado correto, a expressão deve ser avaliada como `float`. Isso pode ser obtido de duas formas:

- adicionando um elemento neutro de tipo `float` à expressão;
- forçando a avaliação de `soma` ou `num` como `float` (*type casting*).

`media.c`

```
#include <stdio.h>  
  
int main ()  
{  
    int soma, num ;  
    float media ;  
  
    soma = 10 ;  
    num = 4 ;  
  
    media = soma / num ;           // errado, perda de precisão  
    printf ("media = %f\n", media) ;  
  
    media = 1.0 * soma / num ;     // soma é "promovida" a float  
    printf ("media = %f\n", media) ;  
  
    media = (float) soma / num ;   // soma é avaliada como float (casting)  
    printf ("media = %f\n", media) ;  
  
    media = soma / (float) num ;   // num é avaliado como float (casting)  
    printf ("media = %f\n", media) ;  
}
```



A avaliação forçada de um valor ou variável para um tipo específico usando o prefixo (`type`), como no exemplo acima, é chamada *type casting*. Essa operação é muito usada, sobretudo na avaliação de ponteiros.

Conversão de ponteiros

Uma operação frequente em C é a conversão de tipos de ponteiros. Muitas funções importantes, como `qsort` e `bsearch`, usam ponteiros genéricos `void*` como parâmetros de entrada, para poder receber dados de diversos tipos.

Como ponteiros para `void` não apontam para nenhum tipo válido de dado, eles não podem ser desreferenciados (ou seja, não é possível acessar diretamente os dados que eles apontam). Por isso, ponteiros para `void` precisam ser convertidos em ponteiros para algum tipo válido antes de serem desreferenciados.

O exemplo a seguir mostra como ponteiros `void*` são convertidos em `int*`, dentro da função `compara_int(a,b)`:

qsort.c

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

#define VETSIZE 10

int vetor[VETSIZE] ;

// compara dois inteiros apontados por "a" e "b"
int compara_int (const void* a, const void* b)
{
    int *pa, *pb ;

    pa = (int*) a ; // "vê" a como int*
    pb = (int*) b ; // idem, b

    if (*pa > *pb) return 1 ;
    if (*pa < *pb) return -1 ;
    return 0 ;
}

int main ()
{
    int i ;

    // preenche o vetor de inteiros com aleatórios
    for (i = 0; i < VETSIZE; i++)
        vetor[i] = random() % 1000 ;

    // escreve o vetor
    for (i = 0; i < VETSIZE; i++)
        printf ("%d ", vetor[i]) ;
    printf ("\n") ;

    // ordena o vetor (man qsort)
    // Protótipo: int (*compara_int) (const void *, const void *)
    qsort (vetor, VETSIZE, sizeof (int), compara_int) ;

    // escreve o vetor
    for (i = 0; i < VETSIZE; i++)
        printf ("%d ", vetor[i]) ;
    printf ("\n") ;
}
```

Conversão de/para strings

No caso específico de strings, a conversão destas para outros tipos é efetuada através de funções específicas:

```
#include <stdlib.h>

// string to float, int, long, long long
double  atof (const char *str) ;
```

```
int      atoi  (const char *str) ;
long     atol  (const char *str) ;
long long atoll (const char *str) ;

// string to double, float, long double, com teste de erro
double strtod (const char *nptr, char **endptr) ;
float  strtof  (const char *nptr, char **endptr) ;
long   strtold (const char *nptr, char **endptr) ;
```

No sentido *número* → *string*, a forma mais simples de converter um dado de qualquer tipo para *string* é usando a função `sprintf`, que formata e “imprime” o dado em uma *string*, de forma similar ao que a função `printf` realiza na saída padrão:

float2string.c

```
#include <stdio.h>

int main ()
{
    char buffer[256] ;
    float x ;

    x = 32.4 / 7 ;

    sprintf (buffer, "%5.4f", x) ; // "imprime" x na string buffer

    printf ("%s\n", buffer) ;

    return 0 ;
}
```

Leitura complementar

- [Type Conversions, C in a Nutshell](#).

From:

<https://wiki.inf.ufpr.br/maziero/> - **Prof. Carlos Maziero**

Permanent link:

https://wiki.inf.ufpr.br/maziero/doku.php?id=c:conversao_de_tipos

Last update: **2023/08/01 17:16**

