PROGRAMAR

REVISTA PORTUGUESA DE PROGRAMAÇÃO • WWW.PORTUGAL-A-PROGRAMAR.PT

EDIÇÃO #50 - SETEMBRO 2015

ISSN 1647-0710

EDIÇÃO 50

O FUTURO DOS BACKENDS?

E AINDA, NINJECT, OPENSHIFT, BITS EM C, WINDOWS 10 IOT E OUTROS MAIS ARTIGOS

A PROGRAMAR

COMO USAR BASE DE DADOS SQLITE EM WINDOWS 10 UNIVERSAL APPS

WINDOWS HELLO A AUTENTICAÇÃO BIOMÉTRICA NO WINDOWS 10

OFFICE GRAPH A INTELIGÊNCIA DO OFFICE 365

RECONHECIMENTO DE VOZ COM JAVASCRIPT

COLUNAS

AS NOVIDADES DO C# 6 **C#**

Manipulação ao nível do bit na Linguagem C

É sabido que um computador trabalha em modo binário, armazenando e manipulando *bits*, isto é, *zeros* e *uns*. Este artigo procura resumir as metodologias mais comuns para uso e manipulação de *bits* através da linguagem C.

Base binária, octal e hexadecimal

A designação *bit* identifica um valor da base binária. Como o nome sugere, a base binária é composta por dois valores distintos, representados por zero e um, daí também se designar por base *dois*. Assim, um *bit* pode assumir um desses dois valores, sendo muitas vezes empregue para representar um estado ativo (*bit* com o valor a 1) ou inativo (*bit* com valor a 0).

Vários *bits* podem ser agrupados para formar valores com maior amplitude. Concretamente, sempre que se acrescenta um *bit*, está-se a duplicar o número de valores passíveis de serem representados pelo conjunto de bits. Por exemplo, com um bit consegue-se representar dois estados (0 e 1). Com dois bits já é possível representar quatro estados (00, 01, 10 e 11) e com três *bits* oito estados (000, 001, 010, 011, 100, 101, 110 e 111). De uma forma geral, *n* bits permitem a representação de 2^n estados diferentes. Por exemplo, 16 bits permitem a representação de 2^{16} valores distintos, isto é, 65536. É essa a razão porque um inteiro de 16 bits sem sinal (i.e., *unsigned*) pode representar valores entre 0 e 65535. Outro exemplo é o do octeto, que designa um conjunto de oito bits e que pode representar 2^8, i.e., 256 valores inteiros, seja entre -128 e 127 (*octeto com sinal*) ou entre 0 e 255 (*octeto sem sinal*). O termo *byte* é frequentemente empregue para designar um octeto.

A representação binária é usualmente pouco conveniente para o ser humano que facilmente se perde na contagem e localização dos *bits*, especialmente se existir um número elevado de *bits*. Deste modo, os programadores usam frequentemente a base octal e a base hexadecimal como alternativa à representação binária. Estas duas bases são empregues por serem mais compactas e pela facilidade com que se consegue converter de uma representação para a representação binária e vice-versa.

A base octal tem um conjunto de oito símbolos para a representação de uma dada quantidade. Os símbolos são os dígitos compreendidos entre 0 e 7. No caso da linguagem C (e de muitas outras), um valor em base octal é representado com um zero à esquerda. Assim, o valor 0123 numa listagem de código C identifica o valor octal 123 e não o valor decimal 123. Na realidade, o valor 0123 corresponde ao valor 83 em base decimal. A conversão de octal para decimal pode ser feita somando-se as parcelas resultantes da multiplicação de 3 por 8^0, de 2 por 8^1 e de 1 por 8^2 obtendo-se o valor 3x1+2x8+8x8=83. Note-se que o uso do zero à esquerda para indicar que uma constante inteira está em base octal pode confundir o programador menos atento que poderá pensar tratar-se de uma constante em base 10 com um insignificante zero à esquerda.

Conforme já referido, o maior interesse da base octal é a facilidade de conversão para a respetiva representação binária e vice-versa. De facto, por ser composta por 8 símbolos, cada símbolo octal é representado em binário por três bits, pois três bits permitem gerar 8 valores distintos ($2^3 = 8$). Por exemplo, o símbolo octal 3 é representado em binário por 011, o símbolo 6 por 110. A Tabela 1 mostra a conversão entre octal e binário para os oito símbolos da base octal. Voltando ao exemplo anterior, o valor octal 123 (0123 na linguagem C) tem a representação binária 001.010.011, correspondendo à substituição dos símbolos da base octal pelos respetivos valores binários (os pontos empregues na representação binária destinam-se somente a simplificar a leitura). Importa referir que algumas linguagens de programação já suportam representação binária, usando o prefixo 0b antes da quantidade numérica. É o caso da linguagem Java (somente a partir da versão 7), na qual o valor binário correspondendo ao valor octal 0123 poderia ser representado como **0b**001010011.

Octal	Binário
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Tabela 1: Mapeamento entre base octal e base binária

De modo similar à base octal, uma representação em base hexadecimal é facilmente convertida numa representação binária e vice-versa. A base hexadecimal assenta num conjunto de 16 símbolos, usando os algarismos 0 a 9 para a representação dos 10 primeiros símbolos e as letras de A a F para os restantes seis símbolos. Nas linguagens de programação, as constantes em base hexadecimal são identificadas pelo prefixo *0x*. Assim, retomando o exemplo anterior, 0x123 representa uma quantidade em base 16. A conversão de uma valor hexadecimal para base 10 consiste em somar as parcelas resultantes da multiplicação do símbolo mais à direita por 16^0 (símbolo designado como o *menos significativo*), da multiplicação do segundo símbolo mais à direita por

16^1, da multiplicação do terceiro símbolo mais à direita por 16^2 e assim sucessivamente. Para o caso do valor 0x123, obtém-se $3x16^0 + 2x16^1 + 1x16^2$, isto é, 3+32+256, ou seja o valor decimal 291. Usando uma notação frequentemente empregue, pode dizer-se que $(123)_{16}$ corresponde ao valor $(291)_{10}$.

A conversão de um valor hexadecimal para a representação binária equivalente processa-se de forma similar à conversão de um valor octal para binário, exceto que cada símbolo hexadecimal deve ser mapeado para um valor de 4 bits de acordo com a Tabela 2. O uso de 4 bits por símbolo decorre do facto que são necessário 4 bits para representar todos os 16 símbolos empregues na base hexadecimal (2^4=16). Aplicando-se a metodologia de conversão hexadecimal para binário ao exemplo 0x123, obtém-se a seguinte representação em binário: 0001.0010.0011.

Hexadecimal	Binário
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Tabela 2: Mapeamento entre hexadecimal e base binária

A maior frequência de uso na programação da representação hexadecimal em relação à representação octal deriva do facto de um valor hexadecimal apresentar um tamanho que é sempre múltiplo de 4 bits. Essa característica possibilita que facilmente possa ser encontrado um valor hexadecimal com o mesmo número de bits de um tipo de dados inteiro. Por exemplo, para o caso de se pretender um valor inteiro com 16 bits, apenas é necessário garantir que a representação hexadecimal tenha 4 símbolos. Similarmente, para um valor de 32 bits, sabe-se que é apropriado um valor hexadecimal com 8 símbolos e assim sucessivamente. Adicionalmente, o formato hexadecimal é empregue para a representação de endereços, dado os endereços terem geralmente um número de bits que é uma potência de dois (8, 16, 32, 64, etc.). Deste modo, não surpreende que a linguagem C disponibilize através da função *printf* e do respetivo operador de formatação *%x*, a representação de um determinado valor em formato hexadecimal. Note-se que em alternativa ao operador %x, pode ser empregue o operador %X (maiúscula) que apresenta o mesmo resultado, exceto que usa maiúsculas para representar os símbolos hexadecimais entre A e F.

Especificação de campos de bits em estruturas

A linguagem C possibilita a declaração de campos binários em estruturas do tipo *struct*. Assim, é possivel declarar um ou mais elementos de uma *struct* como sendo um conjunto de bits. A manipulação do conjunto de bits assim definidos faz-se através do campo da *struct*. Considere-se o exemplo da *struct exemplo1* apresentado na Listagem 1, na qual estão declarados os campos campo01 e campo02, respetivamente com dois e quatro bits. O uso de um campo de bits é efetuado da mesma forma que qualquer outro campo da estrutura, especificando-se o nome do campo. No caso da Listagem 1 são atribuídos os valores 1 (em decimal, correspondendo a 01 em binário) e 0xA (em hexadecimal, correspondendo a 1010 em binário), respetivamente, aos campos *campo01* e *campo02*.

```
/* Exemplo: "bit_fields.c" */
#include <stdio.h>
typedef struct exemplo1{
      int campo_bit01:2;
      unsigned int campo_bit02:4;
      float valor_float;
}exemplo1_t;

exemplo1_t  exemplo;
exemplo1.campo01 = 1;
exemplo1.campo02 = 0xA;
printf("campo01=%d\n", exemplo1.campo01);
printf("campo02=%d\n", exemplo1.campo02);
```

Listagem 1: exemplo bit_fields.c

Importa notar que um elemento especificado como campo de bits deve ser obrigatoriamente declarado como sendo do tipo int (ou equivalentemente do tipo signed int) ou do tipo unsigned int. O número de bits definido para o campo condiciona os valores que lá podem ser armazenado. Assim, para o caso do campo01, os dois bits do campo permitem armazenar um dos conjuntos binários 00, 01, 10 ou 11. Adicionalmente, dado que campo01 é declarado com *int*, isto é, inteiro com sinal, os valores inteiros que o campo pode armazenar são o -2, o -1, 0 e 1. Por sua vez, o elemento campo02 tem espaço para quatro bits, pelo que lhe pode ser atribuído um valor hexadecimal desde que tenha somente um dígito, como é o caso do valor 0xA empregue na Listagem 1.

Bits e variáveis inteiras

Na linguagem C, o acesso ao nível do *bits* não está limitado a campos de bits definidos em *structs*. De facto, é possível efetuar operações envolvendo operadores binários em variáveis do tipo inteiro, sejam elas int, short, long ou mesmo char, independentemente de ser considerado o sinal ou não (signed/unsigned). A principal diferença entre o uso de um campo de bits e o uso de uma variável inteira reside no facto que uma operação binária numa variável inteira envolve todos os bits da variável, ao passo que num campo de

bits, apenas são afetados os bits do campo de *bits*. Assim, quando se efetua uma operação binária envolvendo, por exemplo, uma variável inteira sem sinal com 32 bits, é necessário considerar os efeitos da operação sobre os 32 bits que compõem a variável. É pois importante, quando se faz uso de uma variável inteira, ter em conta o número de bits da variável, algo que pode ser determinado multiplicando o resultado devolvido pelo operador *sizeof* por 8, dado que esse operador devolve o tamanho em octetos (*bytes*) da variável ou do tipo de dados que lhe é passado como parâmetro (Listagem 2). A norma C99 introduziu tipos de dados com tamanho explicitado, como é o caso do tipo int8_t que corresponde a um valor inteiro com sinal de 8 bits (i.e., um octeto) ou o uint16_t que tem 16 bits para guardar valores inteiros sem sinal (Open-STD, 2003). A norma C99 especifica ainda que os tipos inteiros explicitados se encontram definidos no ficheiro <inttypes.h>.

```
int var_a;
printf("nº bits 'int'=%d\n",sizeof(var_a)*8);
printf("nº bits 'short'=%d\n", sizeof(short)*8);
```

Listagem 2: exemplo sizeof.c

Conceito de transbordo

Por terem um número finito de bits, as variáveis do tipo inteiro apenas podem representar um número finito de valores inteiros compreendidos entre um valor mínimo e um valor máximo. Por exemplo, uma variável do tipo `uint16` apenas pode representar os valores inteiros do intervalo [0, 2^{16}-1], isto é, [0, 65535]. Assim, caso se pretenda guardar um valor maior do que aquele suportado pela variável, ocorrerá o que se designa por um transbordo, perdendo-se a parte mais significativa do resultado. A Listagem 3 exemplifica o que sucede quando se soma uma unidade à variável `transbordo_1` do tipo `uint16` (inteiro sem sinal de 16 bits) que foi previamente carregada com o máximo valor que suporta, isto é, 65535: o valor da variável passa para 0. A Listagem 3 mostra ainda o transbordo da variável `transbordo_2` que é do tipo `int16`, isto é, uma variável inteira de 16 bits com sinal, que pode ser empregue para representar os valores do intervalo inteiro [-32768, +32767]. Assim, quando se carrega a variável com o valor máximo (32767) e posteriormente se soma uma unidade à variável, o valor da variável passa a ser o valor mais negativo, isto é, -32768 (ver Listagem 4). A possibilidade de transbordo é algo ao qual o programador deve estar muito atento, pois usualmente provoca comportamentos erráticos da aplicação (Baraniuk, 2015).

```
/*
 * Exemplo: "transbordo.c"
* Compilar:
 * gcc -Wall -W -std=c99 transbordo.c -o transbor-
do.exe
*/
#include <stdio.h>
#include <inttypes.h>

int main(void){
    uint16_t transbordo_1;
    int16_t transbordo_2;
  printf("Nº de bits de 'unsigned short': %u\n",
                          sizeof(transbordo_1)*8);
 transbordo_1 = 65535;/* Carrega valor máximo */
    printf("valor de transbordo_1=%u\n",
                                  transbordo_1);
    transbordo_1++; /* transbordo! */
    printf("(após +1) transbordo_1=%u\n",
                                  transbordo_1);
    transbordo_2 = 32767;
    printf("valor de transbordo_2=%d\n",
                                  transbordo_2);
    transbordo_2++; /* transbordo! */
    printf("(após +1) transbordo_2=%d\n",
                                  transbordo_2);
    return 0;
}
```

Listagem 3: exemplo transbordo.c

```
Nº de bits de 'unsigned short': 16
valor de transbordo_1=65535
(após +1) transbordo_1=0
valor de transbordo_2=32767
(após +1) transbordo_2=-32768
```

Listagem 4: resultados da execução de transbordo.c

Operações binárias acessíveis na linguagem C

Por operação binária entende-se a operação que tem por operando(s) um ou mais valores que são tratados de forma binária, isto é, as operações decorrem bit a bit.

As operações binárias disponibilizadas na linguagem C correspondem às operações habituais de manipulação de bits que são: 1) negação; 2) "e" (conjunção); 3) "ou" (disjunção); 4) "ou exclusivo" (disjunção exclusiva); 5) deslocamento para a esquerda e 6) deslocamento para a direita. As operações binárias são usualmente executadas de forma muito eficiente pelo computador, pois muitos processadores implementam nativamente as operações binárias.

Detalham-se de seguida, as operações binárias anteriormente enumeradas.

Operador de negação

Como o nome sugere, a operação de negação consiste na troca bit a bit, sendo que um bit a 1 é convertido para um bit a 0, e vice-versa. Na linguagem C, a operação de negação (*not* na designação anglo-saxónica) é representada pelo operador ~ (tilde). O operador de negação é dito unário, porque apenas requer um operando. Na Listagem 5, o operador de negação binária é empregue para atribuir à variável out o resultado da negação do conteúdo da variavel in, isto é, a negação de 0x012345678, resultando no valor 0xfedcba98 conforme mostrado na Listagem 6.

```
/* Exemplo: "not_binario.c"*/
#include <stdio.h>

int main(void){
```

```
	unsigned int in = 0x01234567;
	unsigned int out;
	out = ~in;
	printf("in: %x\n", in);
	printf("out: %x\n", out);
	return 0;
}
```

Listagem 5: exemplo not_binario.c

```
	in: 1234567
	out: fedcba98
```

Listagem 6: resultado da execução de not_binario.c

Operador *AND binário*

O operador "e" binário é tambem designado por operador de conjunção. É ainda conhecido pela sua designação anglo□saxónica *and*, sendo identificado na linguagem C através do símbolo "&". O operador requer dois operandos. A tabela de verdade do operador (Tabela 3) mostra que uma operação de AND binário em que pelo menos um dos operandos é o bit 0 resulta sempre no resultado bit 0. Pelo contrário, se um dos operandos for bit a 1, então o resultado corresponderá ao bit do outro operando: será 1 se o bit do outro operando for 1 e 0 se o outro operando for 0. Essas características do *and* binário podem ser empregues para conhecer o valor de um determinado bit (*and* binário com um dos operandos a 1) ou para *zerar* um determinado bit (*and* binário com um dos operandos a 0). A Listagem 7 apresenta código onde é efetuada a operação *and binário* entre os valores numéricos 0x12 (0001.0010 em binário) e 0x0F (0000.1111). A operação produz o resultado binário 0000.0010 (0x02 em hexadecimal), correspondendo ao *and binário* entre cada bit homólogo dos dois operandos 0x12 e 0x0F.

***and* binário (&)**	**0**	**1**
0	0	0
1	0	1

Tabela 3: tabela de verdade do operador and (&)

```
/* Exemplo: "and_binario.c" */
#include <stdio.h>

int main(void){
	int a = 0x12; /* 0001.0010b, 18 base 10 */
	int b = 0x0F; /* 0000.1111b, 15 base 10 */
	int c;
	c = a & b;  /* and binario */
	/* 0001.0010 & 0000.1111 => 0000.0010 */
	printf("c = %d & %d => %x\n", a, b, c);
	return 0;
}
```

Listagem 7: exemplo and_binario.c

É importante distinguir o operador *and binário* do operador *and lógico* no contexto da linguagem C. Em termos de representação, o primeiro é representado pelo símbolo &, ao passo que o operador *and lógico* requer dois símbolos & (&&). No que respeita à funcionalidade, o operador *and lógico* (&&) trata os operandos como entidades lógicas, isto é, tendo um valor verdadeiro ou falso, usualmente designado de *booleano*, e não *bit* a *bit* como sucede com o operador *and binário*. Assim, por exemplo, na expressão if((a==0) && (b==2)){...}, a mesma será considerada verdadeira apenas se o valor da variavel a for 0 **e** se o valor da variável b for 2, isto é, se ambas as operações (a==0) e (b==2) tiverem valor lógico verdadeiro. Se qualquer uma das expressões for falsa, ou ambas, então o resultado do *and lógico* é falso. A Tabela 4 mostra a tabela de verdade do operador *and lógico*.

***and* lógico (&&)**	**Verd.**	**Falso**
Verd.	Verd.	Falso
Falso	Falso	Falso

Tabela 4: tabela de verdade do operador and lógico (&&)

A Listagem 8 efetua a operação de *and lógico* sobre as condições (a==0) e (b==2) atribuíndo o valor resultante da operação à variável inteira *result*. Da análise do resultado da execução do código (Listagem 9), verifica-se que, na linguagem C, o valor lógico verdadeiro é mapeado para o valor inteiro 1 (um), e o valor lógico falso para o valor inteiro 0 (zero). Esse mapeamento mantém-se mesmo com o aparecimento do tipo de dados bool_t com a norma C99.

```
/* Exemplo: "and_logico.c" */
#include <stdio.h>
int main(void){
	int a = 0;
	int b = 2;
	int result;
	/* Condicao verdadeira */
	result = ((a==0) && (b==2));
	printf("verdadeiro => %d\n", result);
	/* Condicao falsa */
	result = ((a==0) && (b==3));
	printf("falso => %d\n", result);
	return 0;
}
```

Listagem 8: exemplo and_logico.c

```
verdadeiro => 1
falso => 0
```

Listagem 9: resultado da execução de and_logico.c

Operador *OR binário*

O operador "ou" binário, corresponde à operação de disjunção. É ainda designado por "OR" binário, sendo representado na linguagem C através do símbolo da barra vertical "|" . A tabela de verdade do operador OR binário (Tabela 5) mostra que sempre que um dos operandos é o bit 1, o resultado final é o bit 1, independentemente do valor do outro operando. Por sua vez, o bit 0 é o elemento neutro do operador OR binário, dado que o resultado de um OR binário com um dos operandos a bit 0 é determinado pelo valor do outro

operando: 0 se o outro operando for bit a 0, e 1 se o outro operando for bit a 1.

Um dos usos do operador OR binário é a ativação de um bit, isto é, colocar a 1 um determinado bit. Por exemplo, o resultado da operação OR binário com o operando 0011, terá sempre os dois bits menos significativos à 1, independentemente do valor do outro operando. De facto, conforme anteriormente observado, sempre que um determinado bit dos operandos do operador OR binário é 1, o resultado do bit correspondente é também ele 1. O código *or_binario.c* (Listagem 10) exemplifica o uso de 0x003, ou seja 0000.0000.0011b, como operando no operador OR binário, originando um resultado cujos dois bits menos significativos têm o valor 1 (Listagem 11).

***or* binário (|)**	**0**	**1**
0	0	1
1	1	1

Tabela 5: tabela de verdade do operador or (|)

```
/* Exemplo: or_binario.c */
#include <stdio.h>

int main(void){
    int a = 0x003;/* 0000.0000.0011b, 3 base10 */
    int b = 0x120;/* 0001.0010.0000b, 288 base10 */
      int c;
      c = a | b;  /* or binario */
      /* 0000.0000.0011 | 0001.0010.0000
      => 0001.0010.0011 */
      printf("c = %d | %d => %d\n", a, b, c);
      return 0;
}
```

Listagem 10: exemplo or_binario.c

```
c = 3 | 288 => 291
```

Listagem 11: resultado da execução de or_binario.c

À semelhança do anteriormente visto para o operador *AND*, existe também na linguagem C um operador *OR lógico*, representado através de dupla barra vertical, isto é, ||. Com exceção da tabela de verdade (Tabela 6) que obviamente difere da tabela do *AND lógico*, tudo o anteriormente mencionado para o operador *AND lógico* se mantém.

***or* lógico (||)**	**Verd.**	**Falso**
Verd.	Verd.	Verd.
Falso	Verd.	Falso

Tabela 6: tabela de verdade do operador or lógico (||)

Operador *ou exclusivo* (xor)

O operador *ou exclusivo, ou xor* na designação anglo-saxónica (contração de *eXclusive OR*) é também conhecido por disjunção exclusiva. Na linguagem C, o operador XOR é representado pelo símbolo **^**. Conforme mostra a tabela de verdade (Tabela 7), a operação de XOR resulta no bit 1 se os dois operandos corresponderem a bits diferentes (i.e., um dos operandos é o bit a 1 e o outro o bit a 0). Caso ambos os operandos representem o mesmo bit, então o resultado da operação de *XOR* é o bit a 0.

***xor* (^)**	**0**	**1**
0	0	1
1	1	0

Tabela 7: tabela de verdade do operador xor (^)

Uma das aplicações do operador XOR binário é o cálculo de paridade de um determinado conjunto de bits. A paridade par de uma sequência de bits diz-se *par* se o número de bits a 1 na sequência é par, e *impar* se o número de bits a 1 na sequência é impar. A Listagem 12 apresenta código em linguagem C que calcula a sequência de paridade de uma sequência de sete inteiros.

```
/* Exemplo: xor_paridade.c */
#include <stdio.h>

int main(void){
      /* Vetor de 7 inteiros sobre os
      quais é calculada a sequencia de paridade */
      int i;
      int paridade; /* Sequência de paridade */
      int vetor_entrada[7];
      vetor_entrada[0] = 0x12;     /* 0001.0010 */
      vetor_entrada[1] = 0x02;     /* 0000.0010 */
      vetor_entrada[2] = 0x22;     /* 0010.0010 */
      vetor_entrada[3] = 0x00;     /* 0000.0000 */
      vetor_entrada[4] = 0xA0;     /* 1010.0000 */
      vetor_entrada[5] = 0xFA;     /* 1111.1010 */
      vetor_entrada[6] = 0x4D;     /* 0100.1101 */
  /* Sequencia de paridade esperada: 0010.0101 */
      paridade = vetor_entrada[0];
      for(i=1; i<7; i++){
           paridade = paridade ^ vetor_entrada[i];
      }
      printf("paridade=0x%x (hex)\n", paridade);
      return 0;
}
```

Listagem 12: exemplo xor_paridade.c

No exemplo apresentado, cada inteiro é representado por dois símbolos hexadecimais, considerando-se assim apenas 8 bits por inteiro (independentemente de cada inteiro ter 32 bits – os bits mais significativos para além do oitavo bit estão a zero). O cálculo da sequência de paridade resume-se a aplicar a operação de XOR de forma iterativa entre os sete elementos da sequência de entrada, usando-se para o efeito a variável paridade para guardar a sequência de paridade. Note-se que a sequência de paridade corresponde à sequência de bits que é necessário acrescentar à sequência de entrada para obter paridade par.

Entrada[i]	Representação binária
[0]	0001.0010
[1]	0000.0010
[2]	0010.0010
[3]	0000.0000
[4]	1010.0000
[5]	1111.1010
[6]	0100.1101
Sequência de paridade	**0010.0101**

Tabela 8: sequência de paridade

É importante observar que no exemplo apresentado se está a considerar as sequências de bits que ocorrem na vertical, calculando-se o respetivo bit de paridade. Por exemplo, uma das sequências é formada pelo bit mais significativo (relembre-se, o bit mais à esquerda) de cada um dos sete valores inteiros, corresponde à sequência 0000.110, sendo o bit de paridade o bit 0 por forma que a sequência de oito bits (sete mais o bit de paridade) tenha paridade par, isto é, um número par de bits a 1. A sequência de paridade é pois 0010.0101, ou equivalentemente, 0x25 em hexadecimal. A Tabela 8 mostra os sete conjuntos de bits e a respetiva sequência de paridade.

Embora a linguagem C não disponibilize o operador lógico XOR, a operação XOR entre valores lógicos pode ser obtida através do recurso aos operadores AND, OR e NOT, conforme mostrado na Equação 1.

a XOR b = (!a && b) || (a && !b) (Eq. 1)

Uma aplicação comum do operador XOR, especialmente em *assembler*, é o de zerar o valor de uma variável. Para o efeito, efetua-se o XOR da variável com ela própria (a = a xor a) levando a que o resultado final seja zero, pois a xor a é forçosamente zero.

Operador *deslocamento para a esquerda*

Como o nome sugere, os operadores de deslocamento efetuam o deslocamento de bits. Na linguagem C, o operador deslocamento para a esquerda tem a seguinte sintaxe: valor << n. O operador de deslocamento à esquerda efetua uma translação em *n* posições dos bits para a esquerda do valor especificado. A Figura 1 ilustra uma operação de deslocamento para a esquerda em um bit do valor 0001.0010, resultando no valor 0010.0100. Note-se que devido ao deslocamento à esquerda em 1 bit, o anterior bit mais significativo (bit mais à esquerda) é perdido, sendo acrescentado um bit a 0 para a posição do bit menos signficativo (bit mais à direita, representado a azul na Figura 1). Caso o deslocamento fosse de *n* bits para a esquerda, perder-se-iam os *n* bits mais significativos, sendo ainda acrescentados *n* bits a 0 como bits menos significativos.

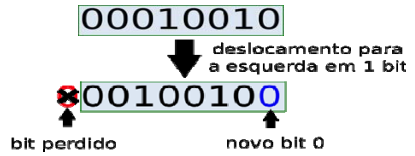


Figura 1: Exemplo de uma operação deslocamento para a esquerda em 1 bit

Quando efetuada sobre a representação binária de um número inteiro, a operação deslocamento para a esquerda em *n* bits produz um resultado final que corresponde à multiplicação por 2^n do valor inteiro original. Por exemplo, na Figura 1, o deslocamento em um bit para a esquerda do valor original 0001.0010 que corresponde ao inteiro 18 em base decimal, é transformado no valor 0010.0100 que corresponde ao valor 36 em base decimal, isto é, ao dobro do valor original. Esta propriedade do operador deslocamento para a esquerda é frequentemente empregue, especialmente em linguagens *assembler*, para efetuar multiplicações de valores inteiros por 2^n, pois é bastante mais rápida do que o algoritmo de multiplicação entre dois números inteiros.

A Listagem 13 exemplifica o uso do operador deslocamento para a esquerda. No exemplo, é aplicado a rotação à esquerda ao valor inteiro 1, usando-se um operando de deslocamento (variável *i*) que incrementa em cada iteração do ciclo *for*. Deste modo, na 1ª iteração do ciclo (i=0), o valor inteiro 1 não é deslocado, não sendo pois alterado. Na iteração seguinte (i=1), o valor 1 é deslocado em 1 bit para a esquerda, passando de 0...001 para 0...010, correspondendo ao valor inteiro 2. Na iteração seguinte (i=2), o valor inteiro 1 é deslocado para a esquerda em dois bits, resultando no valor 0...100, correspondendo ao valor 4. A Listagem 14 apresenta a saída gerada pela execução do programa. Facilmente se depreende que o código da Listagem 13 gera as sucessivas potências do número inteiro 2 (1, 2, 4, 8, 16, 32, 64, 128,...). Acresce-se ainda que os números inteiros potências de dois são frequentemente empregues como operandos dos operadores AND e OR pelo facto da respetiva representação binária comportar apenas um bit a 1, sendo os restantes 0. É frequente a designação de *máscara* para caracterizar um valor inteiro cuja representação binária tenha somente um bit a 1 ou, pelo contrário, somente um bit a 0.

```c
/* Exemplo: shift_left.c */
#include <stdio.h>
int main(void){
      unsigned int valor = 1;
      unsigned int valor_shift;
      size_t size_bits=sizeof(valor)*8;
      unsigned int i;
      for(i=0;i<size_bits;i++){
            valor_shift = valor << i;
            printf("[shift (valor << %02u)]%u\n",
                        i, valor_shift);
      }
      return 0;
}
```

Listagem 13: exemplo shift_left.c

```
[shift (valor << 00)]1
[shift (valor << 01)]2
[shift (valor << 02)]4
[shift (valor << 03)]8
[shift (valor << 04)]16
(…)
[shift (valor << 29)]536870912
[shift (valor << 30)]1073741824
[shift (valor << 31)]2147483648
```

Listagem 14: saída da execução do programa shift_left.c

Operador *deslocamento para a direita*

O operador deslocamento para a direita funciona de forma análoga ao operador de deslocamento para a esquerda, alterando-se somente o sentido do deslocamento. Assim, na operação de deslocamento para a direita em *n* bits, há lugar à deslocação em *n posições dos* bits para a direita. A Figura 2 ilustra uma operação de deslocamento para a direita. Na linguagem C, o operador deslocamento para a direita é representado por **>>**, e à semelhança do operador deslocamento para a esquerda requer dois operandos. Do lado esquerdo do operador fica o operando cujo valor irá ser alvo da operação de deslocamento para a direita. Por sua vez, o operando do lado direito indica de quantos bits deve o valor inicial ser deslocado.

valor_deslocado = valor_inicial >> num_bits;

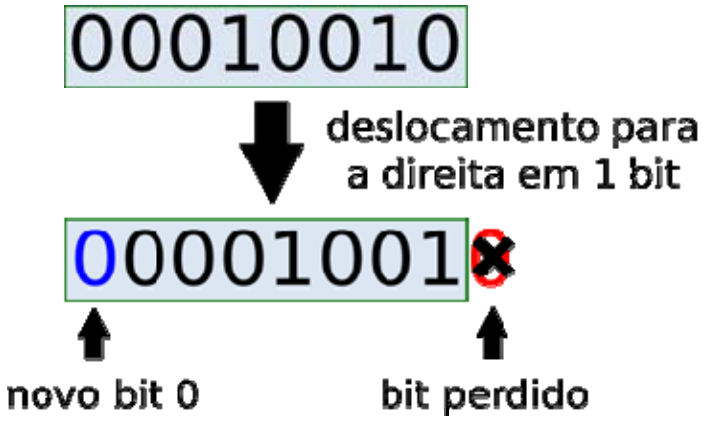


Figura 2: Exemplo de uma operação deslocamento para a direita em 1 bit

```
/* Exemplo: shift_right.c */
int main(void){
      int positive = 998;
      unsigned int sem_sinal = 998;
      int positive_shift_R;
      unsigned int sem_sinal_shift_R;
      int i;
      for(i=0; i < 4; i++){
            positive_shift_R = positive >> i;
            printf("===[i=%d]===\n", i);
            printf("positive_shift_R=%d\n",
                        positive_shift_R);
            sem_sinal_shift_R = sem_sinal >> i;
            printf("sem_sinal_shift_R=%d\n",
                  sem_sinal_shift_R);
      }
      return 0;
}
```

Listagem 15: Exemplo shift_right.c

```
===[i=0]===
  positive_shift_R=998
  sem_sinal_shift_R=998
===[i=1]===
  positive_shift_R=499
  sem_sinal_shift_R=499
===[i=2]===
  positive_shift_R=249
  sem_sinal_shift_R=249
===[i=3]===
  positive_shift_R=124
  sem_sinal_shift_R=124
```

Listagem 16: Saída da execução de right_shift.c

A Listagem 15 exemplifica a operação de deslocamento para a direita em duas variáveis de tipos diferentes. A variável sem_sinal é do tipo unsigned int, isto é, um inteiro sem sinal, ao passo que a variável positive corresponde a um inteiro com sinal (tipo int). Ambas as variáveis são inicializadas com o valor 998, sendo aplicada, sucessivamente, a ambas as variáveis a operação de deslocamento para a direita com 0, 1, 2 e 3 bits de deslocamento. A saída resultante da execução do código é mostrada na Listagem 16. Da análise da saída observa-se que a operação de deslocamento de *n* bits para a direita corresponde à divisão inteira por 2^n do valor inicial. Por exemplo, a operação de deslocamento para a direita em dois bits equivale à divisão inteira por 4 (2^2) do valor inicial. É contudo necessário ter em atenção que se trata de uma divisão inteira, perdendo-se a parte não inteira do resultado e que este comportamente, conforme veremos mais adiante, apenas é válido para operandos do tipo unsigned, isto é, sem sinal. Por exemplo, a divisão de 998 por 8 (2^3) é 124,75, mas quando se procede ao deslocamento em 3 bits para a direita (998 >> 3), obtém-se o valor inteiro 124. Recomenda-se pois cautela no uso do operador deslocamento à direita para efeitos de divisão por 2^n (Steele, 1977).

Uma outra limitação do operador >> envolve o bit mais à esquerda que deve ser acrescentado pelo operador quando ocorre um deslocamento para a direita. De facto, no caso de um valor inteiro representado em complementos de dois, o bit mais à esquerda (bit mais significativo) corresponde ao sinal do número: 0 indica número positivo, ao passo que 1 corresponde a um número negativo. Assim, a operação deslocamento para a direita não pode simplesmente acrescentar um bit zero em lugar do bit mais significativo, pois tal poderá resultar num valor com sinal diferente do valor inicial. No caso da linguagem C, não está definido qual o bit a ser inserido como bit mais significativo pelo operador deslocamento à direita quando lida com inteiros com sinal. Deste modo, o comportamento fica dependente do compilador empregue. No caso do código da Listagem 17, quando compilada com o GCC numa plataforma Linux verifica-se que o operador deslocamento à direita mantém o sinal da valor inicial conforme ilustram os resultados da Listagem 18.

```
/* shift_right_signed.c */
#include <stdio.h>
int main(void){
      int positive = 998;
      int negative = -998;
      int positive_shift, negative_shift;
      int I;
      for(i=0; I < 4; i++){
            printf("===[shift right %d]===\n",i);
            positive_shift = positive >> I;
            negative_shift = negative >> I;
      printf("positive_shift=%
                              d\n",positive_shift);
```

```
      printf("negative_shift=%d\n",negative_shift);
      }
      return 0;
}
```

Listagem 17: Exemplo shift_right_signed.c

```
===[shift right 0]===
positive_shift=998
negative_shift=-998
===[shift right 1]===
positive_shift=499
negative_shift=-499
===[shift right 2]===
positive_shift=249
negative_shift=-250
===[shift right 3]===
positive_shift=124
negative_shift=-125
```

Listagem 18: Saída da execução de right_shift_signed.c

Concretamente, na plataforma considerada, o operador deslocamento para a direita acrescenta um bit a zero se o valor inicial for não negativo, e um bit a um se o valor inicial for negativo. Importa observar, que para números negativos, e considerando que o operador deslocamento à direita aplica a persistência do bit mais significativo, a operação de deslocamento de *n* bits para a direita já não produz uma divisão por 2^n com truncagem. Por exemplo, a operação -998 >> 2 resulta, conforme mostrado na Listagem 18, no valor -250 e não no valor -249 como seria expectável pelo facto da divisão de -998 por 4 resultar em -249,5. Esta particularidade do operador deslocamento à direita tem causado erros em vários sistemas, nomeadamente compiladores conforme discutido por Steele Jr. já em 1977 (Steele, 1977). De modo a evitar da armadilha do operador de deslocamento para a direita, a linguagem Java disponibliza o operador deslocamento para a direita sem sinal, representado pelo símbolo >>>. Esse operador preenche sempre a posição do bit mais significativo com um bit a zero.

Casos de usos

Apresentam-se de seguida alguns dos casos de usos mais frequentes de manipulação binária.

Deteção do estado de um bit

A deteção do estado de um bit consiste em determinar o valor do $i^{ésimo}$ bit de um determinada sequência de bits. Para o efeito, faz-se uso do operador AND binário, tendo como operandos o valor que se pretende analisar e uma máscara binária. A máscara binária é inteiramente composta por bits a zero, exceto para o bit a um na posição do bit cujo valor se pretende detetar.

```
/*
 * Exemplo: mostra representação em
 * bits do valor inteiro da variável valor
 *
*/
#include <stdio.h>
#include <assert.h>

int is_bit_um(int valor, int num_bit){
      int num_bits_int = sizeof(valor) * 8;
      assert( num_bit < num_bits_int );
      int mascara_num_bit = (1 << num_bit);
      return ( valor & mascara_num_bit );
}
int main(void){
      int hex = 0xF0F1F2F3;
      int bit_i, i;
      int total_bits = sizeof(hex) * 8;
      printf("Conversão de 0x%X:\n",hex);
      for(i=total_bits-1;i>=0;i--){
            bit_i = is_bit_um(hex, i) ? 1 : 0;
            printf("%d",bit_i);
            if( (i % 4 == 0) && (i>0)){
                  printf(".");
            }
      }
      printf("\n");
      return 0;
}
```

Listagem 19: Exemplo mostra_em_bin.c

No programa mostra_em_bin.c (Listagem 19), a função is_bit_um devolve zero se o bit num_bit do parâmetro valor é zero e não zero (valor lógico verdadeiro) se o bit num_bit for um. Para o efeito, a função atribui à variável mascara_num_bit um bit a um na posição pretendida através da operação de deslocamento à esquerda, aplicando posteriormente a mascára através da operação de AND binário. No exemplo, a função é chamada sucessivamente para mostrar cada um dos bits da variável hex, disponibilizando assim a representação binária do valor da variável hex como ilustra a Listagem 20.

```
Conversão de 0xF0F1F2F3:
1111.0000.1111.0001.1111.0010.1111.0011
```

Listagem 20: Saída da execução de mostra_em_bin.c

Ativação/desativação seletiva de bits

A ativação seletiva de bits consiste em ativar, num determinado valor inteiro, um ou mais bits a um. Por sua vez, a desativação seletiva de bits corresponde à operação inversa, isto é, colocar um ou mais bits a zero.

A ativação seletiva de bits efetua-se através da operação de OR binário, usando como operandos o valor que se pretende modificar e uma máscara apropriada. A máscara deve ser constituída por bits a zero, exceto para os bits que se pretendem ativar, que devem estar a um. Por exemplo, caso se pretenda ativar os 4 bits menos significativos de um valor inteiro, deve-se empregar como máscara o valor binário que tenha os quatro bits menos significativos a 1111, estando os restantes a zero. Deste modo, e considerando um valor de 16 bits, a máscara corresponderá ao valor 0X000F, ativando-se os 4 bits menos significativos através da operação: novo_valor = valor | 0x000F.

A desativação seletiva de bits é conseguida através da operação de AND binária, usando como operadores, o valor que se pretende alterar e uma apropriada máscara binária. A máscara binária deve ser composta por bits a zero

nas posições que se pretendem desativar e bits a um nas restantes posições. Por exemplo, caso se pretendam desativar os 4 bits menos significativos de um valor de 16 bits, usar-se-á a máscara 0xFFF0, resultando na seguinte operação: novo_valor = valor & 0xFFF0.

Deteção de valores potências de dois

Determinar se o valor de uma determinada variável inteira sem sinal corresponde a uma potência de dois é uma operação trivial quando se recorre a operações binárias. De facto, dado que uma potência de dois tem um e só um bit a um (e.g., 16 que é 0001.0000 em binário), subtraindo-se uma unidade à potência de dois, obtém-se um valor que tem todos os bits à direita do bit ativo da potência de dois a um, e a zero o bit ativo bem como todos os bits à esquerda do bit ativo da potência de dois. Por exemplo, subtraindo uma unidade a 16 obtém-se 15, correspondendo a 0000.1111 em binário. Assim, para determinar se um determinado valor é uma potência de dois, basta efetuar uma operação de AND binário entre o valor e o valor menos uma unidade. Se o resultado for zero, o valor em apreço é uma potência de dois. É importante notar que este algoritmo só é válido para valores positivos. A função is_potencia_dois (Listagem 21) faz uso dessas propriedades das potências de dois para detetar se parâmetro valor corresponde ou não a uma potência de dois. A saída da execução do programa is_potencia_dois é mostrada na Listagem 22.

```
/*
 * Exemplo: operações binária para averiguar
 * se número positivo é potência de dois.
*/
#include <stdio.h>
int is_potencia_dois(unsigned int valor){
      if( valor == 0 || valor == 1 ){
            return 0;
      }
      return ((valor & (valor-1)) == 0? 1:0);
}
int main(void){
      unsigned int i;
      for(i=2;i <= (1<<10); i++){
            if( is_potencia_dois(i) ){
                  printf("%u\n", i);
            }
      }
      return 0;
}
```

Listagem 21: Exemplo is_potencia_dois.c

```
2
4
8
16
32
64
128
256
512
1024
```

Listagem 22: Execução de is_potencia_dois.c

Ativação de opções

Algumas funções da linguagem C requerem o uso da operação de OR binário por forma a que seja possível especificar múltiplas opções através de um parâmetro. Um exemplo é a função open que é empregue para a abertura de um ficheiro. Conforme mostra a Listagem 23, a função apresenta dois parâmetros. O primeiro corresponde ao caminho do ficheiro que se pretende manipular. Mais interessante para o âmbito deste artigo, é o segundo parâmetro, designado de *flags*, pois permite a especificação de vários elementos. De facto, a documentação da função open (e.g., man 2 open num sistema Linux) indica que podem ser especificada, entre outros, constantes para a criação de um ficheiro. Por exemplo, a criação de um ficheiro somente para escrita é especificada através de O_CREAT | O_WRONLY | O_TRUNC, isto é, especificando-se as opções O_CREAT, O_WRONLY e O_TRUNC através do operador OR binário. O valor que é efetivamente recebido pela função open corresponde pois ao resultado da operação de OR binário das três constantes. Na prática, as três constantes são potências de dois, significando que cada uma apenas têm um bit ativo. Tal é confirmado pelo programa open_flag.c (Listagem 24) que mostra o valor numérico das constantes O_CREAT, O_WRONLY e O_TRUNC (Listagem 25). Deste modo, torna-se possível passar, através de um mesmo parâmetro, várias configurações, sendo cada configuração especificada por um ou mais bits. Contudo, é necessário ter em conta que esta metodologia de empacotamento em bits de configurações requer código do lado da função chamada para que essa possa identificar as configurações pretendidas pela função chamante.

```
int open(const char *pathname, int flags);
```

Listagem 23: protótipo da função open

```
/*
 * Mostra o valor numérico de algumas das
 * constantes
 * que podem ser empregues pela função open
 */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
int main(void){
      printf("O_WRONLY = %X\n", O_WRONLY);
      printf("O_CREAT = %X\n", O_CREAT);
      printf("O_TRUNC = %X\n", O_TRUNC);
      return 0;
}
```

Listagem 24: Exemplo open_flags.c

```
O_WRONLY = 1
O_CREAT = 40
O_TRUNC = 200
```

Listagem 25: Saída do programa open_flags.c

Notas finais

A manipulação ao nível do bit é algo que os programadores da linguagem C devem conhecer por forma a tirar o melhor

partido da linguagem. Embora o seu uso explícito seja mais comum na programação sistema de baixo nível, o exemplo da função open ilustra que a manipulação ao nível de bit, embora de forma implícita, ocorre frequentemente na linguagem C.

> ***(...) a base binária é composta por dois valores distintos, representados por zero e um, daí também se designar por base dois (...)***

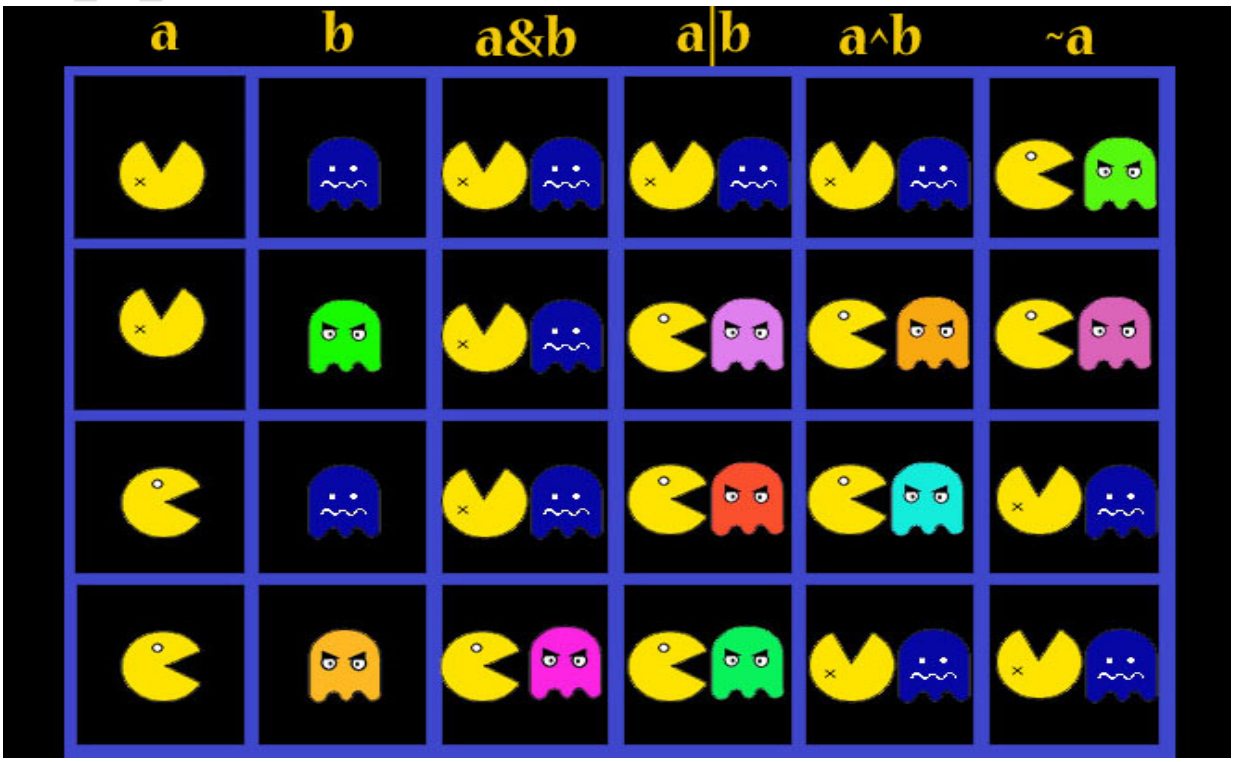
Para quem tem necessidade de recorrer à manipulação ao nível do bit, é ainda importante ter em conta os problemas, uns mais subtis do que outros, que podem ser encontrados. É exemplo disso o uso da operação de deslocamento à direita, cujo comportamento varia consoante o compilador e a plataforma que se está a usar.

Bibliografia

Baraniuk, C. (05 de 05 de 2015). *The number glitch that can lead to catastrophe.* Obtido de BBC: http://www.bbc.com/future/story/20150505-the-numbers-that-lead-to-disaster

Open-STD. (2003). *Rationale for International Standard - Programming Languages - C .* Obtido de http://www.open-std.org/JTC1/SC22/WG14/www/C99RationaleV5.10.pdf

Steele, G. L. (1977). Arithmetic shifting considered harmful. *ACM SIGPLAN Notices, 11*(12), 61-69. Obtido de http://dspace.mit.edu/bitstream/handle/1721.1/6090/AIM-378.pdf



AUTOR

Escrito por Patrício Domingues

doutorado em Engenharia Informática e professor do Departamento de Engª Informática na Escola Superior de Tecnologia e Gestão (ESTG) do Instituto Politécnico de Leiria (IPLeiria). Tem lecionado, entre outras, a disciplina de Programação Avançada da Licenciatura em Engenharia Informática. É ainda responsável pelo GPU Education Center (antigo NVIDIA CUDA Teaching) da ESTG/IPLeiria.