

Capítulo 6

PROGRAMAÇÃO DE BAIXO NÍVEL EM C

6.1 Introdução

Além de ser uma linguagem de alto nível, C é uma linguagem que oferece certas facilidades de baixo nível que permitem ao programador desenvolver programas que seriam possíveis apenas com o uso de *assembly*. Estas facilidades de baixo nível da linguagem C serão apresentadas neste capítulo.

Como as operações apresentadas aqui envolvem manipulações de seqüências de bits que são um tanto incômodas de serem escritas em formato **binário**, usualmente, os formatos **octal** e **hexadecimal** são utilizados para representar estas seqüências de bits. Portanto, antes de prosseguir, é importante que o aluno adquira habilidade de transformar seqüências de bits entre esses três formatos. A **Tabela 20** serve como referência auxiliar para essa tarefa:

Decimal	Binário	Hexadecimal	Octal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	8	10
9	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	D	15
14	1110	E	16
15	1111	F	17

Tabela 1: Conversões entre Bases

Para converter uma seqüência de bits do formato binário para hexadecimal, divida a seqüência dada em seqüências menores de quatro bits, completando (se for o caso) a primeira seqüência de quatro bits com zeros à esquerda. Por exemplo, a seqüência 110101 pode ser dividida em 0011 e 0101 (note que a primeira seqüência foi completada com zeros à esquerda). Após esta divisão, escreva os números hexadecimais correspondentes a cada seqüência de quatro bits utilizando a **Tabela 1**. Na seqüência de bits do exemplo, 0011 corresponde a 3 e 0101 corresponde a 5; portanto, 110101 corresponde a 35 em hexadecimal, que se escreve em C como 0x35 (v. **Seção 1.2.3**). Para converter de hexadecimal para binário, transforme cada dígito hexadecimal numa seqüência de quatro bits utilizando a **Tabela 1**. Por exemplo: 0xA52B corresponde a 1010010100101011, pois A corresponde a 1010, 5 corresponde a 0101, 2 corresponde a 0010, e B corresponde a 1011. As transformações entre octais e binários são feitas de modo semelhante, mas utilizam seqüências de três bits, ao invés de quatro bits. Outras transformações possíveis entre estes formatos de números devem ter sido aprendidas em curso introdutório de computação. Convença-se de que realmente sabe como fazer estas conversões antes de prosseguir.

6.2 Operadores de Manipulação de Bits

Alguns programas requerem a manipulação individual de bits dentro de uma palavra de memória. A linguagem C possui vários operadores que permitem que tais operações sobre bits possam ser executadas. Estes operadores podem ser divididos em três categorias:

- **Operador complemento-de-um;**
- **Operadores lógicos sobre bits;** e
- **Operadores de deslocamento.**

Estes operadores serão discutidos nas seções a seguir.

6.2.1 Operador Complemento-de-um

O operador **complemento-de-um**, representado pelo símbolo `~` (til), é um operador unário cujo efeito é o de inverter os bits de seu operando (i.e., cada bit 1 torna-se 0 e cada bit 0 torna-se 1). Este operador é prefixo, de modo que ele deve sempre preceder seu operando. O operando do complemento um deve ser de um tipo inteiro (**int**, **long**, **short**, ou **char**). Normalmente, este operando é representado por um valor hexadecimal ou octal. Para avaliar o resultado da aplicação do operador complemento-de-um sobre um número decimal, octal ou hexadecimal, siga a seguinte sequência de passos:

1. Transforme o número em binário;
2. Aplique o operador a cada bit nesta última representação; e
3. Finalmente, escreva o resultado em decimal, octal ou hexadecimal novamente.

Suponha, por exemplo, que se deseje avaliar `~0xA52B`. Seguindo a sequência de passos acima, tem-se:

1. `0xA52B` corresponde a 1010010100101011
2. `~0xA52B` é igual a 0101101011101000
3. 0101 1010 1101 0100 corresponde a `0x5AD4`

Portanto, `~0xA52B` é igual a `0x5AD4`.

Exercício: Calcule o valor de `~9430` e escreva o resultado em hexadecimal. (Note que 9430 é um número em formato decimal.)

O operador complemento-de-um é também denominado simplesmente de **operador de complemento**. Este operador faz parte do mesmo grupo de precedência dos outros operadores unários de C e sua associatividade é da direita para a esquerda.

6.2.2 Operadores Lógicos sobre Bits

Existem três operadores lógicos sobre bits:

- **conjunção (e) sobre bits**, representado pelo símbolo `&`;
- **disjunção (ou) sobre bits**, representado pelo símbolo `|`;
- **disjunção exclusiva (ou-exclusivo) sobre bits**, representado pelo símbolo `^`.

Cada um desses operadores requer dois operandos inteiros (**int**, **long**, **short**, ou **char**). As operações são executadas a cada par de bits correspondentes em cada operando (é por isto que estes operadores recebem a qualificação *sobre bits*). Isto é, o primeiro bit do primeiro operando é comparado com o primeiro bit do segundo operando, o segundo bit do primeiro operando é comparado com o segundo bit do segundo operando, e assim por diante até que todos os bits sejam comparados. O resultado desta comparação é apresentado na **Tabela 2**, onde `b1` e `b2` representam um bit do primeiro operando e o bit correspondente do segundo operando, respectivamente.

b1	b2	b1 & b2	b1 b2	b1 ^ b2
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	1	0

Tabela 2: Operações Sobre Bits

Para avaliar o resultado da aplicação de qualquer operador lógico bit-a-bit sobre dois números inteiros decimais, octais ou hexadecimais, siga a seguinte sequência de passos:

- 1. Transforme cada operando em binário;
- 2. Aplique o operador a cada par de bits correspondentes; e
- 3. Finalmente, escreva o resultado em decimal, octal ou hexadecimal novamente.

Por exemplo, suponha que `a` seja igual a `0x6DB7` e `b` seja igual a `0xA726`. Então, as expressões: `a & b`, `a | b`, e `a ^ b` podem ser calculadas como mostrado a seguir:

- 1. `0x6DB7` corresponde a `0110 1101 1011 0111` (a)
`0xA726` corresponde a `1010 0111 0010 0110` (b)
- 2. `a & b` resulta em `0010 0101 0010 0110`
`a | b` resulta em `1110 1111 1011 0111`
`a ^ b` resulta em `1100 1010 1001 0001`
- 3. `0010 0101 0010 0110` corresponde a `0x2526`
`1110 1111 1011 0111` corresponde a `0xEFB7`
`1100 1010 1001 0001` corresponde a `0xCA91`

Portanto, `0x6DB7 & 0xA726` é igual a `0x2526`, `0x6DB7 | 0xA726` é igual a `0xEFB7`, e `0x6DB7 ^ 0xA726` é igual a `0xCA91`.

Cada operador lógico sobre bits tem sua própria precedência. O operador `&` tem a maior precedência, depois vem o operador `^` e, finalmente, o operador `|`, que tem a menor precedência. O operador `&` está numa classe de precedência imediatamente abaixo daquela dos operadores de igualdade (`==` e `!=`). O operador `|` está numa classe de precedência imediatamente acima daquela do operador `&&`. Todos estes operadores têm associatividade da esquerda para a direita. O **Apêndice B** apresenta um quadro completo de operadores da linguagem C.

Note que apesar de compartilharem o mesmo símbolo `&`, não existe possibilidade de ambigüidade entre os operador de endereço e o operador de conjunção sobre bits, pois o primeiro é unário, enquanto o segundo é binário. Neste capítulo, o símbolo `&` refere-se sempre ao operador de conjunção sobre bits.

É importante notar ainda que apesar de apresentarem alguma semelhança com os operadores lógicos (inclusive na denominação), os operadores lógicos *sobre bits* apresentados aqui não devem ser confundidos com os operadores lógicos apresentados na Seção 1.4. Na verdade, estes dois conjuntos de operadores possuem mais diferenças dos que semelhanças entre si e esta confusão freqüentemente acarreta em erro. Por exemplo, um erro bastante freqüente entre os iniciantes na linguagem C é usar o operador `&` no lugar do operador `&&`. A seguir são enumeradas as principais diferenças entre os operadores lógicos e os operadores lógico sobre bits:

- Os operadores lógicos podem ser aplicados a operandos de quaisquer tipos primitivos de C, enquanto que os operadores lógicos sobre bits podem ser aplicados apenas a valores dos tipos inteiros.
- A aplicação de operadores lógicos considera os valores integrais de seus operandos, enquanto que, no caso dos operadores lógicos sobre bits, são considerados os bits que compõem os operandos. Por exemplo, os resultados das expressões a seguir ilustram este fato:

`2 && 12` resulta em `1`

```
2 & 12 resulta em 0

2 || 12 resulta em 1
2 | 12 resulta em 14

! 2 resulta em 0
~ 2 resulta em -3
```

- Os operadores `&&` e `||` possuem ordem de avaliação de operandos definida, enquanto que isto não ocorre com os operadores `&` e `|`.
- Uma expressão lógica contendo o operador `&&` ou `||` pode ser parcialmente avaliada, enquanto que isto não ocorre com nenhum operador lógico sobre bits. Por exemplo, o trecho de programa a seguir:

```
int x = 0;
if (x && 10/x)
    ...
```

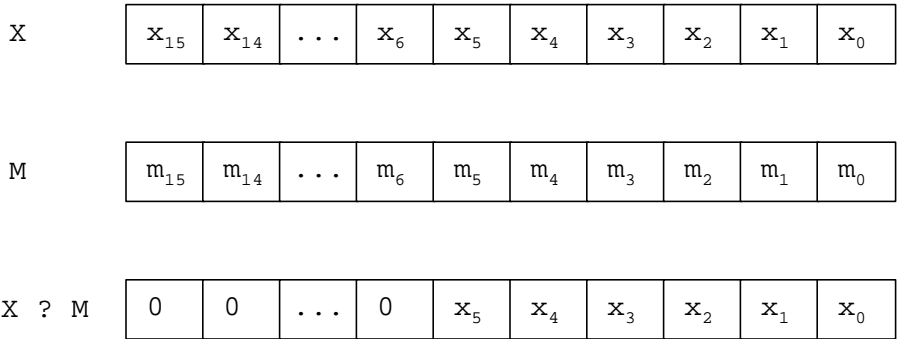
funciona perfeitamente bem, mesmo que a expressão `10/x` represente, neste caso, uma divisão por zero. No entanto, se o operador `&&` for substituído por `&`, ocorrerá um erro em tempo de execução.

6.2.3 Mascaram

Mascaram é uma operação muito utilizada em programação na qual uma dada seqüência de bits é transformada numa seqüência desejada por meio de uma operação lógica sobre bits. Nesta operação, a seqüência dada é um dos operandos do operador sobre bits e o outro operando é uma seqüência denominada de **máscara**. A máscara e a operação lógica são escolhidas de modo que a operação de mascaram resulte na seqüência desejada.

Existem vários tipos de operações de mascaram. Por exemplo, parte de uma palavra de memória pode ser copiada para uma outra palavra, enquanto que o restante da nova palavra é preenchida com zeros. Esta operação é demonstrada a seguir.

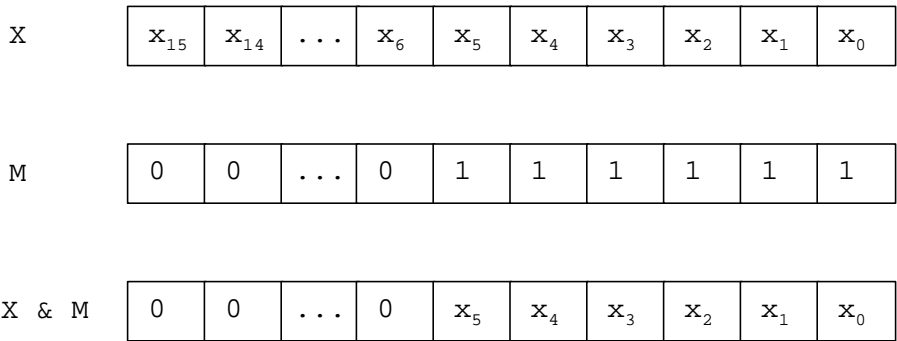
Suponha que `x` seja uma variável do tipo **unsigned int**, e que se deseje extrair os seis bits mais à direita de `x` e atribuí-los a uma variável `y`. Os bits restantes de `y` devem ser preenchidos com zeros. A questão é: como escolher o operador lógico sobre bits e a máscara para esta operação de mascaram? Para uma melhor visualização do problema, a operação de mascaram é ilustrada nos diagramas a seguir (suponha que o tipo **int** ocupe 16 bits de memória):



Note que nos diagramas acima, `x` é uma variável qualquer do tipo **int** (conseqüentemente, os bits `xi` de `x` também são arbitrários). A máscara `M` (mais precisamente, os bits de `M`) deve ser escolhida adequadamente, de modo a resultar na seqüência de bits dada por `x ? M`, onde `?` representa um operador sobre bits também a ser convenientemente escolhido¹. Examinando-se a tabela dos operadores sobre bits apresentada na **Seção 6.2.2**, pode-se concluir que o operador deve ser **&**, e que a máscara deve ser constituída por uma

¹ Aqui, é feita a suposição de que o tipo **int** ocupa 16 bits, mas isto não é uma limitação.

seqüência de zeros, correspondente à porção do resultado contendo apenas zeros (i.e., os bits numerados de 6 a 15), e uma seqüência de uns, correspondente à porção do resultado contendo os valores originais do operando dado (i.e., os bits numerados de 0 a 5). Levando isso em consideração, os diagramas podem ser reapresentados como:



Portanto, a máscara da operação de mascaragem do exemplo acima é a constante com representação binária: 0000 0000 0011 1111. A representação hexadecimal desta máscara é 0x3F.

É interessante notar ainda que a máscara do último exemplo é independente do tamanho do tipo `int` utilizado na operação, pois seus bits mais à esquerda são todos zeros. Por exemplo, se o tipo `int` ocupasse 32 ou 64 bits, a máscara continuaria sendo 0x3F (a única diferença seria que a representação binária conteria mais zeros, mas o valor da máscara seria o mesmo). Entretanto, máscaras cujos bits mais à esquerda são iguais a 1 não são independentes do tamanho do tipo utilizado na operação de mascaragem. Por exemplo, suponha que se deseje uma operação de mascaragem semelhante à do último exemplo, mas agora com a extração dos seis bits mais à esquerda (e não à direita) de `x`, e a atribuição de zeros aos bits restantes à direita (e não à esquerda). Se você entendeu o exemplo acima, vai verificar facilmente que o operador sobre bits continuará a ser `&`, mas, neste caso, a máscara terá a representação binária: 1111 1100 0000 0000. A representação hexadecimal desta máscara é 0xFC00, e esta máscara agora depende do tamanho do tipo `int`. Por exemplo, se o tamanho do tipo `int` fosse 32 bits, ao invés de 16 bits, sua representação binária seria:

1111 1111 1111 1111 1111 1100 0000 0000

o que corresponde a 0xFFFFFC00 em hexadecimal (verifique isto). Portanto, a máscara deste último exemplo é dependente de implementação (lembre-se que o tamanho do tipo `int` depende do compilador e da máquina utilizados). Felizmente, existe uma maneira de remover esta dependência: escrevendo a máscara em termos de seu complemento. Quando aplica-se o operador complemento-de-um a um número, os zeros transformam-se em uns, e os uns transformam-se em zeros. Portanto, se um número possui 1 em sua posição mais à esquerda, seu complemento terá 0 nesta mesma posição. Assim, a máscara tornar-se-á independente de implementação. Utilizando esta estratégia a operação de mascaragem não mais será representada por `X & M`, mas sim por `X & ~M`. Ou seja, a operação será: `X & ~(complemento de M)`². Prosseguindo, o complemento da máscara do último exemplo é dado por: 0x3FF, e este valor é independente de implementação (verifique isto). Finalmente, a operação de mascaragem pode ser apresentada como:

X & ~0x3FF

onde `x` é um valor qualquer do tipo `int`.

Como um outro exemplo de operação de mascaragem, suponha que se deseje copiar uma porção de uma seqüência de bits de uma palavra para uma nova palavra, com o restante da nova palavra sendo preenchida com uns. Neste caso, o operador *ou bit-a-bit* (`|`) é utilizado e a máscara deve conter uns nas posições correspondentes aos bits que se deseja que sejam uns no resultado e zeros nas posições correspondentes aos bits da seqüência dada que devem ser preservados na nova palavra. Conforme visto antes, e a porção de uns

² Isto decorre do fato de `~~M` ser igual a `M`, pois invertendo-se cada bit de `M` duas vezes, obtém-se o valor original de cada bit (verifique isto).

correspondem aos bits mais à esquerda do resultado, a máscara será dependente do tamanho da palavra utilizada, de modo que para tornar a operação de mascaragem independente de implementação, deve-se utilizar o complemento da máscara como foi feito no último exemplo.

Exercício: Escreva uma operação de mascaragem, independente de implementação, para copiar os seis bits mais à direita de um valor do tipo **int** para uma nova palavra, com os bits restantes mais à esquerda sendo preenchidos com uns.

Como último exemplo de operação de mascaragem, uma sequência de bits pode ser copiada para uma nova palavra, com os bits restantes sendo invertidos e copiados para a nova palavra. Esta operação de mascaragem pode ser efetuada com o uso do operador de disjunção exclusiva (^). Por exemplo, suponha que se tenha um valor do tipo **int** do qual se deseje copiar os 8 bits mais à esquerda numa variável do tipo **int**, enquanto que os 8 bits mais à direita deste valor são copiados invertidos na variável. Neste caso, o operador a ser utilizado será o *ou-exclusivo bit-a-bit*, e a máscara deverá ter a forma binária:

0000 0000 1111 1111

É fácil verificar que essas escolhas para operador e máscara produzirão o resultado desejado. De fato, quando um dos operandos do operador ^ é 0, o bit resultante será aquele correspondente ao outro operador. Portanto, os oito bits mais à esquerda do resultado da operação terão os valores originais do operando dado. Por outro lado, quando, um dos operandos do operador ^ é 1, o resultado da operação é igual ao outro operando invertido (v. **Tabela 2**). Finalmente, o valor hexadecimal desta máscara é 0xFF (confira!) e este valor é independente de implementação (por que?). Se fosse desejado preservar os 8 bits mais à direita e inverter os 8 bits mais à esquerda, a operação de mascaragem deveria ser escrita em termos do operador de complemento para tornar-se independente de implementação (verifique isto).

O operador ou exclusivo sobre bits pode também ser utilizado como um *alternador* que modifica o valor de um bit particular de uma palavra de memória alternadamente entre 0 e 1. Isto é, se um bit particular tem valor 1, ele passará a ser 0 após a operação, e vice-versa. Suponha, por exemplo, que se deseje inverter o valor do terceiro bit, a partir da direita, de um valor do tipo **int**. Então, a máscara a ser utilizada em tal operação consiste de uma sequência de zeros, com exceção do terceiro bit mais à direita que terá o valor 1; ou seja, o valor binário da máscara será: 0000 0000 0000 0100, que corresponde a 0x4 em hexadecimal.

Exercício: Suponha que o número a ser alternado tenha o valor hexadecimal 0x6DB7. Mostre que sucessivas aplicações da máscara 0x4 e do operador ^ alternarão o resultado da operação de mascaragem entres os valores 0x6DB7 e 0x6DB3, com o terceiro bit mais à direita sendo alternado entre 0 e 1.

Operações de alternância do tipo desse último exemplo são comuns em manipulação de baixo nível de hardware.

6.2.4 Operadores de Deslocamento

Existem dois **operadores de deslocamento** de bits em C: **deslocamento à esquerda**, representado pelo símbolo <<, e **deslocamento à direita**, representado pelo símbolo >>. Cada um destes operadores requer dois operandos inteiros: o primeiro operando representa uma sequência de bits a ser deslocada, enquanto que o segundo operando representa o número de deslocamentos que o primeiro operando irá sofrer (i.e., de quanto o primeiro operando será deslocado). O valor do segundo operando não poderá ser maior do que o número de bits utilizados para representar o primeiro operando.

A operação de deslocamento à esquerda faz com que os bits do primeiro operando sejam deslocados para a esquerda um número de posições indicado pelo segundo operando. Os

bits mais à esquerda do primeiro operando, em número igual ao valor do segundo operando, serão perdidos na operação, enquanto que os bits mais à direita do primeiro operando, também em número igual ao valor do segundo operando, serão preenchidos com 0. Por exemplo, suponha que se deseje calcular o resultado da operação $0x6DB7 \ll 6$. Então, o primeiro passo é transformar o primeiro operando em binário:

$0x6DB7$ é representado por 0110 1101 1011 0111 em binário.

Então, deslocam-se os bits desta representação binária seis posições para a esquerda, resultando em:

0110 1101 1100 0000

Este último número binário representa $0x6DC0$ em hexadecimal. Portanto, $0x6DB7 \ll 6$ é igual a $0x6DC0$. Note que os seis bits mais à esquerda do número original foram perdidos e que os seis bits mais à direita do mesmo número foram preenchidos com zeros.

A operação de deslocamento à direita é similar à operação de deslocamento à esquerda quando o primeiro operando é **unsigned** (v. adiante), mas agora os bits são deslocados para a direita. Por exemplo, suponha que se deseje calcular o resultado da operação $0x6DB7 \gg 6$. Então, o primeiro passo é transformar o primeiro número em binário:

$0x6DB7$ é representado por 0110 1101 1011 0111 em binário.

Então, deslocam-se seis posições para a direita os bits desta representação binária, resultando em:

0000 0001 1011 0110

Este último número binário representa $0x1B6$ em hexadecimal. Portanto, $0x6DB7 \gg 6$ é igual a $0x1B6$. Note que os seis bits mais à direita do número original foram perdidos e que os seis bits mais à esquerda do mesmo número foram preenchidos com zeros³.

Se o primeiro operando do operador de deslocamento à direita for um número negativo, o resultado do deslocamento é dependente de implementação. Isto é, alguns compiladores preenchem as posições deslocadas à esquerda com zeros, como visto acima, enquanto outros compiladores preenchem estas posições com uns. Em outras palavras, alguns compiladores preenchem as posições à esquerda com o valor do bit mais à esquerda (i.e., o bit de sinal), enquanto que outros compiladores preenchem estas posições com zeros independentemente do bit de sinal.

É importante observar que deslocar um número inteiro sem sinal para a esquerda é equivalente a multiplicá-lo pela potência de dois do deslocamento. Isto é,

$$x \ll y \text{ é equivalente a } x * 2^y$$

Entretanto, esta relação deixa de ser válida quando x é **signed** e o deslocamento faz com que x mude de sinal. Por exemplo, $5 \ll 1$ é equivalente a $5 * 2^1$, que é igual a 10, mas $1 \ll 15$ é igual a -2^{15} , e, portanto, a relação não é válida neste caso.

De modo semelhante, deslocar inteiros não-negativos para a direita é equivalente a dividi-lo pela potência de dois do deslocamento. Isto é,

$$x \gg y \text{ é equivalente a } x / 2^y$$

³ Compare este último exemplo com o exemplo anterior de deslocamento à esquerda. Note que os operandos são os mesmos em ambos os exemplos.

Por exemplo, `255 >> 3` é equivalente a $255 / 2^3$, que é igual a 31.

Quando o primeiro operando é **unsigned**, estas equivalências são seguramente verdadeiras e as operações de deslocamento são mais eficientes do que as operações aritméticas equivalentes.

É importante observar ainda que qualquer das operações de deslocamento apresentará um comportamento imprevisível quando o segundo operando é negativo ou maior do que o tamanho do tipo do valor deslocado na implementação utilizada.

6.2.5 Operadores de Atribuição sobre Bits

Como ocorre com os operadores aritméticos, a linguagem C oferece operadores que representam as operações sobre bits, vistas acima, combinadas com atribuição em operações únicas. Estas operações são representadas pelos **operadores de atribuição bit-a-bit** apresentados na **Tabela 3**.

Operador	Expressão	Expressão Equivalente
<code>&=</code>	<code>a &= b</code>	<code>a = a & b</code>
<code> =</code>	<code>a = b</code>	<code>a = a b</code>
<code>^=</code>	<code>a ^= b</code>	<code>a = a ^ b</code>
<code><<=</code>	<code>a <<= b</code>	<code>a = a << b</code>
<code>>>=</code>	<code>a >>= b</code>	<code>a = a >> b</code>

Tabela 3: Operadores de Atribuição Bit-a-bit

Os operadores de atribuição bit-a-bit têm a mesma precedência e a mesma associatividade que os outros operadores de atribuição apresentados na **Seção 1.3**. Além disso, o uso destes operadores é recomendado em situações semelhantes às aquelas recomendadas para os operadores de atribuição aritmética (novamente, v. **Seção 1.3**).

6.2.6 Representação Interna de um Valor Inteiro

Nesta seção, será apresentado um exemplo prático de manipulação de bits em C. O exemplo consiste de uma função, denominada `ApresentaRepresentacaoInterna()`, que imprime na tela do computador a sequência de bits correspondente à representação interna de um valor do tipo **int** passado como parâmetro para a função.

```
#include <limits.h>
/****
 *
 * ApresentaRepresentacaoInterna()
 *
 * Apresenta no meio de saída a representação interna de um parâmetro
 * do tipo int.
 *
 ****/
void ApresentaRepresentacaoInterna(int numero)
{
    int i, bit, numeroDeBits;
    unsigned int mascara; /* A máscara deve ser unsigned neste exemplo */

    numeroDeBits = CHAR_BIT*sizeof(int); /* Cada byte contém CHAR_BIT bits */

    mascara = 0x1 << (numeroDeBits - 1); /* Coloca 1 na posição mais à
                                           /* esquerda da máscara. Todos
                                           /* os outros bits terão 0s.

    for (i = 1; i <= numeroDeBits; i++) { /* Imprime cada bit a partir da
        bit = (numero & mascara) ? 1 : 0; /* esquerda do número (parâmetro)
    }
```



```
printf("%x", bit);
if (i % 4 == 0) /* Separa bits em seqüências de quatro bits para */
    printf(" "); /* melhorar a visualização da representação */

mascara >>= 1; /* Move bit com valor 1 da máscara */
/* uma posição para a direita */
}
}
```

A função `ApresentaRepresentacaoInterna()` apresentada acima poderia tornar-se mais geral de modo a imprimir a representação interna de um parâmetro de qualquer tipo (e não apenas do tipo `int`). Para isto, a função deveria receber como parâmetros um ponteiro genérico (i.e., do tipo `void *`) apontando para o objeto que se deseja apresentar a representação interna, e o tamanho do objeto apontado pelo primeiro parâmetro. Em outras palavras, esta nova versão da função `ApresentaRepresentacaoInterna()` deveria ter como protótipo:

```
void ApresentaRepresentacaoInterna2( const void *ptrObjeto,
                                     size_t tamanhoDoObjeto );
```

Exercício: Implemente esta nova versão da função `ApresentaRepresentacaoInterna2()`. Por que é necessário utilizar o tamanho do objeto como parâmetro nesta nova versão?

6.3 Operadores de Memória

Operadores de memória são operadores utilizados em referências a posições e conteúdos de memória. Todos estes operadores já foram exaustivamente discutidos anteriormente, e serão incluídos aqui apenas por uma questão de complemento e para facilidade de referência.

Os operadores de memória de C juntamente com seus significados são apresentados na **Tabela 4**.

Operador	Nome	Uso	Interpretação
<code>&</code>	endereço de	<code>&x</code>	O endereço da variável <code>x</code> .
<code>*</code>	referência	<code>*p</code>	Valor do objeto armazenado na posição apontada por <code>p</code> .
<code>[]</code>	elemento de arranjo	<code>a[i]</code>	Valor do elemento do arranjo <code>a</code> de índice <code>i</code> .
<code>•</code>	ponto	<code>e.c</code>	Valor do campo <code>c</code> da estrutura <code>e</code> .
<code>-></code>	seta à direita	<code>p->c</code>	Valor do campo <code>c</code> da estrutura apontada por <code>p</code> .

Tabela 4: Operadores de Memória

Os operadores `[]`, `->` e `•` (juntamente com o operador de chamada de função `()`) fazem parte de um mesmo grupo de precedência. Este grupo tem a maior precedência dentre todos os operadores da linguagem C. A associatividade destes operadores é da esquerda para a direita. Os operadores unários `&` e `*` fazem parte de um grupo que segue imediatamente o grupo dos operadores `[]`, `()`, `->` e `•` em ordem de prioridade. A associatividade dos operadores `&` e `*` é da direita para a esquerda.

Este capítulo conclui o estudo de todos os operadores da linguagem C. O **Apêndice B** apresenta um resumo de todos estes operadores juntamente com a precedência e associatividade de cada um deles.

6.4 Campos de Bits

Em alguns programas, precisa-se trabalhar com tipos de dados que necessitam apenas de uns poucos bits de memória para serem armazenados. Por exemplo, uma variável que sinaliza uma condição verdadeira/falsa (ou sim/não) precisa de apenas um bit de armazenamento; uma variável que armazena um valor representando um dia do mês, precisa de apenas 5 bits. A linguagem C permite que tais variáveis, que requerem pouco espaço de armazenamento, sejam acomodadas numa única palavra de memória, tornando, assim, o programa mais econômico em termos de armazenamento.

O armazenamento de várias variáveis numa única palavra é possível com o uso de **campos de bits** que são armazenados como campos de uma estrutura. Um campo de bits pode ser acessado como um campo qualquer de uma estrutura comum. A principal diferença entre um campo de bits e um campo comum de uma estrutura está na forma de declaração do campo de bits. Na declaração de um campo de bits, o tipo e o nome do campo devem ser seguidos por dois pontos (**:**) e pelo tamanho (i.e., o número de bits) do campo. O tamanho do campo não pode ultrapassar o tamanho do tipo **int** na implementação utilizada. Segundo o padrão ANSI/ISO, o tipo de cada campo de bits deve ser **int** (**signed** ou **unsigned**), embora muitos compiladores permitam também que este tipo seja **short** ou **char**. Por exemplo, a declaração de estrutura a seguir:

```
struct {  
    unsigned int  a : 1;  
    int          b : 4;  
    int          c : 3;  
    unsigned int  d : 2;  
} E;
```

declara uma estrutura **E** com quatro campos de bits: **a**, **b**, **c**, e **d**, com tamanhos (em bits) dados por 1, 4, 3, e 2, respectivamente. Portanto, estes campos juntos ocupam um total de 10 bits. Se o tamanho da palavra do computador utilizado for maior do que este valor (por exemplo, 16 ou 32 bits), os bits restantes na palavra não serão utilizados. Por outro lado, se a palavra do computador for de 8 bits, duas palavras serão necessárias para armazenar esta estrutura, sendo que apenas dois bits de uma destas palavras será efetivamente utilizado. Note a economia de memória obtida com o uso de campos de bits no exemplo acima. Suponha que o tamanho da palavra (e do tipo **int**) seja 16 bits. Então, se os campos de bits da estrutura **E** fossem declarados como variáveis comuns ou como campos comuns de uma estrutura, seriam necessários 48 bits ($= 8 + 16 + 8 + 16$). Com o uso de campos de bits, apenas uma palavra (i.e., 16 bits) é suficiente para armazenar estes campos; portanto, aqui é feita uma economia de 32 ($= 48 - 16$) bits. Obviamente, se há apenas uma ou umas poucas estruturas deste tipo num programa, a economia de memória é irrelevante, mas se houver um arranjo de milhares de estruturas deste tipo, a economia poderá ser substancial.

Conforme já foi dito, campos de bits podem ser acessados do mesmo modo que campos comuns de estruturas. No entanto, algumas restrições, que não se aplicam a campos comuns, são aplicadas ao uso de campos de bits. Estas restrições são as seguintes:

- O operador de endereço (**&**) não pode ser utilizado com um campo de bits;
- Não se pode acessar um campo de bits por meio de ponteiro; e
- Uma função não pode retornar um campo de bits.

Um campo de bits não pode ultrapassar os limites de uma palavra em memória. Em outras palavras, não se pode ter parte de um campo de bits numa palavra e outra parte em outra palavra. Portanto, se a soma dos bits dos campos de bits acomodados numa palavra, e que antecedem um dado campo de bits, mais o tamanho deste campo exceder o tamanho de uma palavra, este último campo de bits será completamente acomodado na próxima palavra. Por exemplo, suponha que o tamanho da palavra do computador utilizado seja 16 bits e que se tenha a seguinte declaração de estrutura:

```
struct {  
    unsigned int  a : 4;  
    int          b : 6;  
    int          c : 5;  
    unsigned int  d : 7;
```

```
} E2;
```

Neste último exemplo, *a*, *b*, e *c* são acomodados numa mesma palavra, visto que a soma de seus tamanhos (15 bits) não excede o tamanho de uma palavra. Entretanto, resta apenas um bit vago nesta palavra para acomodar o campo *d*. Como este campo tem tamanho igual a 7 bits, o espaço vago nesta palavra é insuficiente para acomodar o campo *d*. Portanto, os 7 bits do campo *d* serão acomodados em outra palavra. Isto é, não pode ocorrer que um bit de *d* seja acomodado na primeira palavra, e os seis bits restantes na segunda palavra, pois um campo de bits não pode ocupar duas palavras. Concluindo o exemplo, os campos *a*, *b*, e *c* ocuparão uma palavra, que terá um bit não utilizado e o campo *d* ocupará outra palavra, que terá 9 bits não utilizados.

Campos de bits podem ser **anônimos**. Um campo de bits anônimo, obviamente, não pode ser referenciado, e seu uso restringe-se ao preenchimento de palavras de modo a controlar o alinhamento de campos de estruturas. Por exemplo:

```
struct {
    unsigned a : 5;      /* Primeira palavra começa aqui */
    unsigned b : 5;
    unsigned : 6;        /* Preenche a primeira palavra */
    unsigned c : 5;      /* Segunda palavra começa aqui */
} E3;
```

Neste último exemplo, o terceiro campo de bits da estrutura *E3* é anônimo, e serve apenas para forçar o armazenamento do campo *c* em outra palavra (i.e., o objetivo do campo de bits anônimo é *alinhar* o campo *c*). Outra forma de controlar o alinhamento de campos de bits é o uso de um campo de bits anônimo de comprimento 0, como por exemplo:

```
struct {
    unsigned a : 5;      /* Primeira palavra começa aqui */
    unsigned b : 5;
    unsigned : 0;        /* Força o campo seguinte a ser */
                          /* alinhado em outra palavra */
    unsigned c : 5;      /* Segunda palavra começa aqui */
} E4;
```

O uso de um campo de bits anônimo de comprimento 0 força o campo que o segue a ser iniciado em outra palavra.

O alinhamento de variáveis, no qual campos de bits anônimos são utilizados, é exigido por alguns processadores. Por exemplo, alguns processadores *Motorola* utilizados na linha de computadores *Macintosh*, requerem que variáveis cujos tamanhos sejam maiores do que 8 bits sejam alinhados em endereços pares em memória. Alinhamento de variáveis com campos de bits anônimos pode resultar em programas não-portáteis. Pior ainda, um programa deste tipo pode funcionar bem numa máquina e não funcionar corretamente em outra. Existem maneiras de se contornarem estes problemas, mas um estudo mais profundo do uso de campos de bits com a finalidade de alinhamento está além do escopo deste livro.

6.5 Aplicações Práticas

6.5.1 Uso de Flags

Suponha que um programa deve tomar uma decisão baseada em qual das setas de direção do teclado está pressionada. Supondo ainda que qualquer destas teclas pode estar pressionada ou não independentemente das outras, nota-se que existem 16 possibilidades de combinações de pressionamento destas teclas. Se você representar o estado (pressionada ou não-pressionada) de cada tecla como uma variável do tipo **unsigned char**, o que parece ser a escolha mais óbvia, seu programa teria o seguinte aspecto:

```
#define PRESSIONADA      1
#define NAO_PRESSIONADA 0
...
unsigned char teclaD, /* Tecla direita */
              teclaE, /* Tecla esquerda */
              teclaC, /* Tecla para-cima */
              teclaB; /* Tecla para-baixo */
...
if (teclaD && teclaE && teclaC && teclaB)
    /* Ação quando as quatro teclas estão pressionadas */
else if (teclaD && !teclaE && !teclaC && !teclaB)
    /* Ação quando apenas a tecla direita está pressionada */
...
else if (!teclaD && !teclaE && !teclaC && !teclaB)
    /* Ação quando nenhuma tecla está pressionada */
```

Portanto, o trecho do programa que toma decisões baseadas no estado de pressionamento das teclas teria um aninhado de ifs que testariam as 16 combinações possíveis. Mas, é possível reduzir a complexidade deste programa.

Em primeiro lugar, note que as variáveis `teclaD`, `teclaE`, `teclaC` e `teclaB` são variáveis flags; isto é, variáveis que, em cada instante, assumem apenas um dentre dois possíveis valores. Este tipo de variável pode ser representado por um único bit e, no exemplo dado, as quatro variáveis podem ser acomodadas numa única variável do tipo **unsigned char**. Resta ainda especificar como determinar o estado das teclas num dado instante.

Este último problema consiste em utilizar constantes que especifiquem o pressionamento de cada tecla isoladamente e então utilizar operadores lógicos sobre bits para determinar o estado das teclas em conjunto. Para tornar as coisas mais palpáveis, suponha que você decida representar as quatro flags de seu programa nos bits 0, 1, 2 e 3 da variável do tipo **unsigned char** que representará o estado das quatro teclas. Então, provavelmente, você começaria a escrever seu programa assim:

```
#define TECLA_D 0x1 /* Tecla direita ocupa bit 0 */
#define TECLA_E 0x2 /* Tecla esquerda ocupa bit 1 */
#define TECLA_C 0x4 /* Tecla para-cima ocupa bit 2 */
#define TECLA_B 0x8 /* Tecla para-baixo ocupa bit 3 */
...
unsigned char estadoDasTeclas;
...
```

Na porção de programa acima, note que o valor da constante que define o pressionamento de uma tecla é dado pela potência de dois da posição ocupada pela respectiva flag na variável `estadoDasTeclas`, que representa o estado de pressionamento de todas as teclas em conjunto. Agora, pode-se determinar se uma determinada tecla está pressionada considerando-se a conjunção sobre bits da variável `estadoDasTeclas` com a constante que representa a tecla desejada. Por exemplo, suponha que se deseje saber se a tecla esquerda está pressionada. Então, neste caso, a situação poderia ser descrita conforme esquematizado a seguir:

estadoDasTeclas	b ₇	b ₆	b ₅	b ₄	b ₃	b ₂	?	b ₀
TECLA_E	0	0	0	0	0	0	1	0
estadoDasTeclas & TECLA_E	0	0	0	0	0	0	?	0

Portanto, conforme pode-se verificar, o resultado da operação `estadoDasTeclas & TECLA_E` é determinado apenas pelo valor do bit 1 da variável `estadoDasTeclas` (representado por "?" na ilustração). Isto é, se o bit 1 for 0, o resultado será 0 e se o valor deste bit for 1, o valor será diferente de zero (o valor preciso do resultado não interessa no contexto atual). Assim, pode-se determinar se a tecla esquerda está pressionada utilizando um teste como:

```

if (estadoDasTeclas & TECLA_E)
    /* Ação quando a tecla esquerda está pressionada */
else
    /* Ação quando a tecla esquerda NÃO está pressionada */

```

O fato de duas ou mais teclas estarem pressionadas pode ser expresso pela disjunção sobre bits (`|`) das constantes que representam as teclas respectivas. Por exemplo, o fato de as teclas direita e esquerda estarem pressionadas pode ser expresso por:

```
TECLA_D | TECLA_E
```

Isto pode não parecer intuitivo à primeira vista. Afinal, o uso do operador de conjunção sobre bits parece ser mais a escolha mais adequada. Entretanto, este seria o caso apenas se as constantes que representam a pressão das teclas fossem bits. Mas, na situação apresentada aqui, as teclas são representadas por valores inteiros cujos bits são todos, exceto um deles, iguais a 0. Além disso, os bits iguais a 1 ocupam posições mutuamente exclusivas nas constantes; isto é, numa constante, este bit ocupa a posição 0, em outra ocupa a posição 1, etc. Portanto, se for feita uma operação de conjunção sobre bits entre quaisquer duas destas constantes, o resultado será zero, indicando que nenhuma das duas teclas respectivas está pressionada, o que não corresponde ao resultado desejado.

Voltando ao exemplo do início desta seção, o trecho de programa ali apresentado poderia ser substituído como mostrado esquematicamente a seguir:

```

#define TECLA_D 0x1 /* Tecla direita ocupa bit 0 */
#define TECLA_E 0x2 /* Tecla esquerda ocupa bit 1 */
#define TECLA_C 0x4 /* Tecla para-cima ocupa bit 2 */
#define TECLA_B 0x8 /* Tecla para-baixo ocupa bit 3 */
...
unsigned char estadoDasTeclas;
...
switch(estadoDasTeclas) {
    case TECLA_D | TECLA_E | TECLA_C | TECLA_B:
        /* Ação quando as quatro teclas estão pressionadas */
    case TECLA_D | TECLA_E:
        /* Ação quando as teclas direita e esquerda são pressionadas */
    case TECLA_D:
        /* Ação quando apenas a tecla direita está pressionada */
        ...
    default:
        /* Ação quando nenhuma tecla está pressionada */
}

```

Portanto, conforme demonstrado neste último trecho de programa, o aninhado de ifs apresentado no início desta seção pode ser substituído por uma instrução **switch** que, conforme foi visto anteriormente (v. Seção 1.5.4.3), é mais legível do que aquele aninhado de ifs.

Caso você ainda esteja convencido da melhor conveniência desta última solução para o problema proposto no início desta seção, suponha que a tomada de decisão quanto à ação a ser seguida de acordo com o estado das teclas seja implementada por uma função. Esta função, evidentemente, precisaria receber da porção do programa que a chama informação sobre o estado das teclas. Então, no caso em que o estado das teclas é representado por variáveis independentes, esta função deveria ter quatro parâmetros, um para cada tecla. Entretanto, no caso em que o estado das teclas é representado por uma única variável, apenas um parâmetro seria necessário.

Se, depois de toda a argumentação apresentada acima você ainda não estiver convencido da estratégia utilizada para representação de flags, faça uma extrapolação. Suponha agora que você precisa trabalhar com trinta flags⁴. Neste caso, uma função que levasse em

⁴ Se você considera este número exageradamente irreal, saiba que uma estrutura que representa uma janela de um sistema de janelas (como Windows, por exemplo) pode possuir cerca de quarenta atributos (por exemplo, estilo da borda, título, tipos de botões, etc.) e muitos destes atributos são

consideração todos os estados das flags teria, no mínimo, trinta parâmetros se a estratégia de uso de flags delineada aqui não fosse adotada.

A seguir serão descritas as operações mais comuns sobre variáveis que armazenam um conjunto de flags.

Ligando uma Flag

Pode-se **ligar** uma flag (i.e., tornar seu valor igual a 1) armazenada numa variável que contém um conjunto de flags, utilizando o operador `|`. Por exemplo, considerando o caso apresentado no início desta seção, suponha que se deseje ligar a flag que representa a tecla esquerda (independentemente do fato de a mesma já estar ligada ou não). Então, a operação seria esquematicamente representada como a seguir:

estadoDasTeclas	b ₇	b ₆	b ₅	b ₄	b ₃	b ₂	?	b ₀
TECLA_E	0	0	0	0	0	0	1	0
estadoDasTeclas TECLA_E	b ₇	b ₆	b ₅	b ₄	b ₃	b ₂	1	b ₀

Observe que, conforme mostra a ilustração anterior, ao final da operação `estadoDasTeclas | TECLA_E`, o bit correspondente à tecla esquerda estará ligado, independentemente de seu valor inicial (fato indicado por "?" na ilustração). Note ainda que os demais bits da variável `estadoDasTeclas` não são modificados pela operação.

Em resumo, a função a seguir pode ser utilizada para ligar uma flag, representada pelo argumento `aFlag`, armazenada num valor do tipo **unsigned long** contendo um conjunto de flags representado pelo argumento `conjuntoDeFlags`.

```
unsigned long LigaFlag( unsigned long conjuntoDeFlags,
                        unsigned long aFlag )
{
    return conjuntoDeFlags | aFlag;
}
```

Desligando uma Flag

Pode-se **desligar** uma flag (i.e., tornar seu valor igual a 0) armazenada numa variável que contém um conjunto de flags, utilizando os operadores `~` e `&`. Por exemplo, considerando o caso apresentado no início desta seção, suponha que se deseje desligar a flag que representa a tecla esquerda (independentemente do fato de a mesma já estar desligada ou não). Então, a operação seria esquematicamente representada como a seguir:

estadoDasTeclas	b ₇	b ₆	b ₅	b ₄	b ₃	b ₂	?	b ₀
TECLA_E	0	0	0	0	0	0	1	0
estadoDasTeclas & ~TECLA_E	b ₇	b ₆	b ₅	b ₄	b ₃	b ₂	0	b ₀

Observe que, conforme mostra a ilustração anterior, ao final da operação `estadoDasTeclas & ~TECLA_E`, o bit correspondente à tecla esquerda estará desligado, independentemente de seu valor inicial (fato indicado por "?" na figura). Note ainda que, novamente, os demais bits da variável `estadoDasTeclas` não são modificados pela operação.

representados por flags (por exemplo, se a janela possui botão OK ou não, se ela é modal ou não, etc.).

Resumindo, a função a seguir pode ser utilizada para desligar uma flag, representada pelo argumento `aFlag`, armazenada num valor do tipo **unsigned long** contendo um conjunto de flags representado pelo argumento `conjuntoDeFlags`.

```
unsigned long DesligaFlag( unsigned long conjuntoDeFlags,
                          unsigned long aFlag )
{
    return conjuntoDeFlags & ~aFlag;
}
```

Invertendo uma Flag

Pode-se **inverter** uma flag (i.e., trocar seu valor) armazenada numa variável que contém um conjunto de flags, utilizando o operador `^`. Por exemplo, considerando o caso apresentado no início desta seção, suponha que se deseje inverter a flag que representa a tecla esquerda (independentemente de seu valor corrente). Esta operação seria esquematicamente representada como a seguir:

estadoDasTeclas	b ₇	b ₆	b ₅	b ₄	b ₃	b ₂	?	b ₀
TECLA_E	0	0	0	0	0	0	1	0
estadoDasTeclas ^ TECLA_E	b ₇	b ₆	b ₅	b ₄	b ₃	b ₂	~?	b ₀

Na ilustração anterior, "?" representa 0 ou 1; conseqüentemente, "~?" irá representar 1 ou 0, respectivamente. É fácil verificar que o resultado realmente é aquele apresentado na ilustração. Suponha que ? seja 1; então ? ^ 1 resultará em 0 e o bit será realmente invertido. Por outro lado, se ? for 0, então ? ^ 1 resultará em 1 e, novamente, o bit estará invertido. Note ainda que os demais bits da variável `estadoDasTeclas` não são modificados pela operação.

A função a seguir pode ser utilizada para inverter uma flag, representada pelo argumento `aFlag`, armazenada num valor do tipo **unsigned long** contendo um conjunto de flags representado pelo argumento `conjuntoDeFlags`.

```
unsigned long InverteFlag( unsigned long conjuntoDeFlags,
                          unsigned long aFlag )
{
    return conjuntoDeFlags ^ aFlag;
}
```

Testando uma Flag

A última operação comum com flags armazenadas numa variável é **testar** se uma flag está ligada ou não. Esta operação já foi suficientemente discutida no início desta seção. Para complementar a discussão, é apresentada uma função que pode ser utilizada para testar se uma flag, representada pelo argumento `aFlag`, armazenada num valor do tipo **unsigned long** contendo um conjunto de flags, representado pelo argumento `conjuntoDeFlags`, está ligada ou não.

```
unsigned int TestaFlag( unsigned long conjuntoDeFlags,
                      unsigned long aFlag )
{
    return conjuntoDeFlags & aFlag;
}
```

6.5.2 Criptografia XOR

O operador `^`, também denominado operador xor⁵, tem propriedades interessantes:

⁵ A denominação *xor* vem do Inglês *exclusive or*; i.e., ou-exclusivo.

1. $x \wedge x$ resulta em zero, qualquer que seja o operando x .
2. $(x \wedge y) \wedge y$ resulta sempre em x . Isto significa que fazer a operação xor de um número inteiro y com o resultado da operação xor de y com outro número inteiro x resulta neste número x .
3. $x \wedge y$ é o mesmo que $y \wedge x$. Ou seja, o operador \wedge é comutativo.

Estas propriedades são fáceis de demonstrar formalmente (faça isso), mas o que interessa aqui são algumas consequências práticas destas propriedades.

Uma consequência da primeira propriedade é que pode-se atribuir 0 a uma variável inteira atribuindo-lhe o resultado da operação xor da variável consigo mesma. Ou seja, utilizando o operador de atribuição $\wedge=$, pode-se atribuir zero a uma variável x através da instrução⁶:

```
x ^= x;
```

A primeira aplicação da segunda propriedade de xor apresentada acima é que ela permite trocar os valores de duas variáveis inteiras sem a utilização de variável auxiliar, como é o caso no algoritmo tradicional. Isto é, se x e y são as variáveis inteiras cujos valores serão permutados, então, o algoritmo que realiza a permuta consiste nos seguintes passos:

1. Faça x receber $x \wedge y$.
2. Faça y receber $x \wedge y$.
3. Faça x receber $x \wedge y$.

É fácil verificar, tomando por base a segunda propriedade do operador xor apresentada, que este algoritmo realmente faz a permuta de valores entre as variáveis x e y . No primeiro passo do algoritmo, a variável x recebe o resultado da operação xor entre x e y . No segundo passo, a variável y também recebe o resultado da operação xor entre x e y , mas agora x representa o valor $x \wedge y$ calculado no passo 1, onde x que aparece nesta última expressão representa o valor original desta variável. Portanto, neste passo está-se na verdade calculando a expressão $(x \wedge y) \wedge y$, onde x representa o valor original de x e, de acordo com a propriedade de xor vista acima, y estará recebendo x (valor original). No terceiro passo, x recebe o valor da expressão $x \wedge y$, mas, como no passo anterior, novamente x aqui representa o valor $x \wedge y$, onde x nesta última expressão representa o valor original de x . Agora, conforme foi visto, no passo 2, y recebeu o valor original de x e, portanto, a expressão $x \wedge y$ no passo 3 significa $(x \wedge y) \wedge x$, onde x nesta última expressão representa o valor original de x . Portanto, utilizando as propriedades 2 e 3 do operador xor, o resultado desta última expressão é y . Assim, neste passo x recebe o valor de y .

Utilizando o operador de atribuição $\wedge=$, o algoritmo pode ser implementado como na seguinte função:

```
void TrocaInteiros(int *x, int *y)
{
    *x ^= *y;
    *y ^= *x;
    *x ^= *y;
}
```

Uma outra aplicação prática das propriedades do operador xor (e bem mais interessante do que as aplicações apresentadas anteriormente) é a implementação de um método para cifrar arquivos denominado **criptografia xor**⁷. Existem várias variantes deste tipo de criptografia,

⁶ Isto não significa, entretanto, que esta forma de atribuição é preferida com relação à forma tradicional. Isto é, a atribuição $x = 0;$ é muito mais legível pois não requer conhecimento profundo sobre a linguagem C. Em resumo, este fato é apresentado aqui mais por uma questão de curiosidade do que por uma questão prática.

⁷ A criptografia consiste de um conjunto de técnicas utilizadas para cifrar arquivos de modo a evitar que pessoas não autorizadas tenham acesso aos conteúdos destes arquivos. A necessidade de segurança cada vez maior devido ao crescente fluxo de informações em redes de computadores tem estimulado o surgimento de algoritmos de criptografia cada vez mais sofisticados. A criptografia xor apresentada aqui é um método de criptografia considerado fraco, mas serve de introdução ao tema.

mas a idéia original é simples e fácil de ser entendida. O programa que faz a criptografia xor recebe como entrada o arquivo a ser criptografado e um caractere (a **chave criptográfica**). Então, o programa executa uma operação xor sobre cada byte no arquivo com a chave criptográfica. Em consequência da segunda propriedade de xor apresentada acima, para decifrar o arquivo assim criptografado, basta executar a mesma operação sobre este arquivo utilizando a mesma chave utilizada para criptografá-lo. O programa a seguir ilustra esta técnica de criptografia.

```
#include <stdio.h>

/****
 *
 * Criptografa()
 *
 * Faz criptografia xor do arquivo cujo nome é passado como primeiro
 * argumento utilizando a chave passada como segundo argumento.
 *
 * Retorno: 1, se a operação for bem sucedida e 0 em caso contrário
 *
 ****/
unsigned Criptografa(const char *arquivo, char chave)
{
    char c;
    FILE *ptrArquivo, /* Stream associado ao arquivo original */
          *ptrTemp;    /* Stream associado a um arquivo temporário */

    /* Abre arquivo original para leitura e gravação no modo binário */
    ptrArquivo = fopen(arquivo, "r+b");

    if (!ptrArquivo)
        return 0; /* Arquivo não pode ser aberto */

    ptrTemp = tmpfile(); /* Cria arquivo temporário */

    if (!ptrTemp) {
        fclose(ptrArquivo); /* Arquivo temporário não foi aberto */
        return 0;
    }

    /* Enquanto o final do arquivo original não for atingido, */
    /* lê cada byte neste arquivo, faz xor do byte lido      */
    /* com a chave e grava no arquivo temporário             */
    while (1) {
        c = getc(ptrArquivo);
        if (feof(ptrArquivo)) /* Testa se final do arquivo */
            break;           /* de entrada foi atingido    */
        putc(c ^ chave, ptrTemp);
    }

    /* Antes de copiar o conteúdo criptografado do          */
    /* arquivo temporário para o arquivo original,          */
    /* é necessário voltar ao início de cada arquivo       */
    rewind(ptrArquivo);
    rewind(ptrTemp);

    /* Copia conteúdo do arquivo temporário para o arquivo original */
    while (1) {
        c = getc(ptrTemp);
        if (feof(ptrTemp)) /* Testa se final do arquivo */
            break;         /* temporário foi atingido    */
        putc(c, ptrArquivo);
    }

    /* Fecha arquivos */
    fclose(ptrArquivo);
    fclose(ptrTemp); /* Arquivo temporário automaticamente destruído */

    return 1;
}
```

```
int main(int argc, char **argv)
{
    unsigned char aChave;

    /* Este programa funciona apenas quando      */
    /* recebe um nome de arquivo como argumento */
    if (argc != 2) {
        printf("Erro: nome de arquivo ausente.");
        return 1;
    }

    printf("\nIntroduza a chave: ");
    aChave = getchar();

    if ( !Criptografa(argv[1], aChave) )
        printf("Erro tentando criptografar arquivo");

    return 0;
}
```

O funcionamento da função que executa a criptografia xor é bastante simples. Esta função recebe o nome do arquivo a ser criptografado e a chave criptográfica como entrada. O arquivo recebido como entrada é aberto para leitura e gravação no modo binário⁸. A função utiliza ainda um arquivo temporário para armazenar temporariamente o resultado da criptografia. Este arquivo é criado e automaticamente aberto no modo `wb+` através da instrução:

```
ptrTemp = tmpfile();
```

Este arquivo é automaticamente destruído quando é fechado utilizando a função `fclose()`⁹.

A criptografia propriamente dita é feita através do laço:

```
while (1) {
    c = getc(ptrArquivo);
    if (feof(ptrArquivo))
        break;
    putc(c ^ chave, ptrTemp);
}
```

que lê cada byte no arquivo original, executa uma operação xor do byte lido com a chave fornecida e grava o resultado da operação no arquivo temporário. Após processar todo o conteúdo do arquivo original, a função deve copiar o conteúdo criptografado que foi gravado no arquivo temporário de volta no arquivo original. Antes disso, porém, é necessário retornar ao início de cada arquivo. Isto é feito através de chamadas da função `rewind()`. Outros detalhes de funcionamento da função `Criptografa()` e da função `main()` que a utiliza são descritos como comentários no próprio programa.

Claramente, a criptografia implementada pelo programa do exemplo anterior é fraca, pois o arquivo criptografado pode ser decifrado em no máximo 256 tentativas¹⁰, que corresponde ao número de valores possíveis para o tipo `char`, que é o tipo do valor utilizado como chave criptográfica.

Exercício: Escreva um programa capaz de descobrir a chave criptográfica de um arquivo criptografado conforme descrito acima sabendo que o arquivo original é um arquivo de texto. (**Sugestão:** Seu programa pode utilizar um laço de repetição aonde os possíveis valores de chave são utilizados para tentar decifrar alguns dos primeiros caracteres do arquivo. A cada passagem no laço, o programa apresentaria estes caracteres supostamente

⁸ Embora o maior interesse seja a criptografia de arquivos-texto, o arquivo é aberto no modo binário, visto que não há interesse aqui em fazer interpretação de conteúdo. Além disso, a função também poderá ser utilizada para criptografar arquivos binários.

⁹ O arquivo temporário também é destruído automaticamente quando o programa é encerrado.

¹⁰ Obviamente, esta estimativa leva em consideração o fato de se saber de antemão como o arquivo foi criptografado (i.e., usando xor e apenas um caractere como chave).

decifrados ao usuário e perguntaria ao mesmo se a tradução faria sentido. Em caso afirmativo, o programa pararia e apresentaria a chave corrente como sendo a chave criptográfica correta; caso contrário, as tentativas do programa prosseguiriam.)

Uma forma de tornar o arquivo criptografado mais difícil de decifrar, ainda utilizando a metodologia básica de criptografia xor, seria utilizar um string, ao invés de um único caractere, como chave criptográfica. A nova versão da função de criptografia, denominada `Criptografa2()`, que implementa esta idéia é apresentada a seguir.

```

/****
 *
 * Criptografa2()
 *
 * Faz criptografia xor do arquivo cujo nome é passado como primeiro
 * argumento utilizando a chave passada como segundo argumento. A
 * diferença com relação à função Criptografa() apresentada antes
 * é que a função atual recebe um string como chave, ao invés de
 * um único caractere.
 *
 * Retorno: 1, se a operação for bem sucedida e 0 em caso contrário
 *
 ****/
unsigned Criptografa2(const char *arquivo, const char *chave)
{
    char          *sequenciaBytes, c;
    size_t        tamanhoChave, nBytesLidos;
    unsigned       i;
    FILE           *ptrArquivo, /* Stream associado ao arquivo original */
                  *ptrTemp;     /* Stream associado a um arquivo temporário */

    /* Abre arquivo original para leitura e gravação no modo binário */
    ptrArquivo = fopen(arquivo, "r+b");

    if (!ptrArquivo)
        return 0; /* Arquivo não pode ser aberto */

    ptrTemp = tmpfile(); /* Cria arquivo temporário */

    if (!ptrTemp) {
        fclose(ptrArquivo); /* Arquivo temporário não pode ser aberto */
        return 0;
    }

    /* O arquivo será lido em quantidades de bytes iguais */
    /* ao tamanho da chave. O arranjo sequenciaBytes */
    /* será utilizado para esta finalidade. */
    tamanhoChave = strlen(chave);
    sequenciaBytes = malloc(tamanhoChave);

    if (!sequenciaBytes)
        return 0;

    /* Enquanto o final do arquivo original não for atingido, lê */
    /* seqüências de bytes do tamanho da chave neste arquivo, faz */
    /* xor da seqüência lida com a chave, byte a byte, e grava no */
    /* arquivo temporário. */
    do {
        nBytesLidos = fread(sequenciaBytes, 1, tamanhoChave, ptrArquivo);

        for (i = 0; i < nBytesLidos; i++)
            sequenciaBytes[i] ^= chave[i];

        fwrite(sequenciaBytes, 1, nBytesLidos, ptrTemp);
    } while ( nBytesLidos == tamanhoChave );

    /* É necessário voltar ao início de cada arquivo */
    rewind(ptrArquivo);
    rewind(ptrTemp);

    /* Copia conteúdo do arquivo temporário para o arquivo original */
    while (1) {
        c = getc(ptrTemp);
        if (feof(ptrTemp)) /* Testa se final do arquivo */
            break; /* temporário foi atingido */
        putc(c, ptrArquivo);
    }

    /* Fecha arquivos */
    fclose(ptrArquivo);

```

```
fclose(ptrTemp); /* Arquivo temporário é automaticamente destruído */

free(sequenciaBytes); /* É necessário liberar o espaço alocado */

return 1;
}
```

A principal mudança apresentada nesta última versão da função de criptografia xor com relação à versão anterior mais simples é o laço **do-while** que executa a criptografia propriamente dita. Neste laço, são lidas seqüências de bytes do tamanho da chave e executadas operações xor entre os bytes da seqüência lida e os bytes correspondentes da chave fornecida. Então, o resultado desta operação é gravado no arquivo temporário. O laço encerra quando é lido um número de bytes menor do que o comprimento da chave fornecida, o que ocorre quando o final do arquivo original é atingido. As partes restantes desta função são semelhantes àquelas da primeira versão apresentada anteriormente e são auto-explicativas.

6.5.3 Endereçamento IP

Um **endereço IP** (*Internet Protocol*) identifica de maneira única um nó ou um host de uma rede IP, e consiste de uma número de 32 bits usualmente divididos em quatro campos de 8 bits (**octetos**), cada qual no intervalo de 0 a 255, separados por pontos, como por exemplo:

150.221.18.7

Este número é algumas vezes representado em forma binária, como por exemplo:

10010110.11011101.00010010.00000111

Um endereço IP consiste de duas partes: uma identifica a rede a outra identifica o nó. A **classe** de um endereço determina que parte do endereço pertence à rede e que parte pertence ao nó. Existem cinco classes diferentes de endereços distinguidas pelo valor do primeiro octeto, conforme mostrado na tabela a seguir:

PRIMEIRO OCTETO DO ENDEREÇO ENTRE ...	CLASSE
1 e 126	A
128 e 191	B
192 e 223	C
224 e 239	D
240 e 255	E

Sabendo a que classe pertence um dado endereço, as partes pertencentes à rede (R) e ao nó (N) são assim determinadas¹¹:

- Class A: RRRRRRRR.NNNNNNNN.NNNNNNNN.NNNNNNNN
- Class B: RRRRRRRR.RRRRRRRR.NNNNNNNN.NNNNNNNN
- Class C: RRRRRRRR.RRRRRRRR.RRRRRRRR.NNNNNNNN

Por exemplo, 150.221.18.7 é um endereço da classe B e, portanto, os dois primeiros octetos identificam a rede e os dois últimos octetos identificam o nó.

Para atribuir um endereço IP a uma rede, a seção correspondente ao nó é especificada com zero em todos seus bits. Por exemplo, 150.221.0.0 identifica a rede do endereço do exemplo anterior.

Muitas vezes, uma rede possui sub-redes derivadas. Avaliando-se a conjunção sobre bits entre uma máscara de sub-rede e um endereço IP, pode-se identificar as seções da rede e do

¹¹ Apenas as classe A, B e C são de interesse daqui em diante.

nó do endereço. As máscaras padrões das classes A, B e C são apresentadas, em formatos binário e decimal, na tabela a seguir:

CLASSE	MÁSCARA PADRÃO (DECIMAL)	MÁSCARA PADRÃO (BINÁRIO)
A	255.0.0.0	11111111.00000000.00000000.00000000
B	255.255.0.0	11111111.11111111.00000000.00000000
C	255.255.255.0	11111111.11111111.11111111.00000000

Note que a máscara padrão de cada classe é composta de uns nos bits correspondentes à parte de endereço de rede e de zeros nos bits correspondentes ao nó. Assim, quando uma operação de conjunção sobre bits entre uma máscara de sub-rede e um endereço IP é executada, o resultado define o endereço da sub-rede. Por exemplo:

Endereço IP:150.215.017.009

Máscara:255.255.000.000

Endereço IP & Máscara:150.215.000.000

Ou, em formato binário:

Endereço IP:10010110.11010111.00010001.00001001

Máscara:11111111.11111111.00000000.00000000

Endereço IP & Máscara:10010110.11010111.00000000.00000000

Uma operação similar é utilizada por pacotes de informação para decidir se dois nós (origem e destino de dados) estão na mesma sub-rede¹². Para verificar se dois nós estão numa mesma sub-rede, executa-se uma conjunção sobre bits dos endereços dos dois nós com a máscara da sub-rede. Se o resultado for o mesmo, os dois nós estão na mesma sub-rede; caso contrário, eles estão em sub-redes diferentes. Por exemplo, dados os endereços 192.158.0.6 e 192.158.1.34, e a máscara de sub-rede 255.255.254.0, tem-se:

Endereço IP:192.158.000.6

Máscara:255.255.254.0

Endereço IP & Máscara:192.158.000.0 (endereço da rede)

Endereço IP:192.158.001.34

Máscara:255.255.254.00

Endereço IP & Máscara:192.158.000.00 (endereço da rede)

Portanto, os endereços 192.158.0.6 e 192.158.1.34 estão na mesma sub-rede. Por outro lado, considerando uma máscara de sub-rede igual a 255.255.254.0 e os mesmos endereços do caso anterior, tem-se:

Endereço:192.158.000.6

Máscara:255.255.255.0

Endereço & Máscara:192.158.000.0 (endereço da rede)

Endereço:192.158.001.34

Máscara:255.255.255.00

Endereço & Máscara:192.158.001.00 (endereço da rede)

Portanto, neste último caso, os nós estão em sub-redes diferentes.

6.6 Portabilidade

(In Progress)

¹² Se os nós estiverem em redes diferentes, o pacote enviado precisará ser roteado para outra rede; caso contrário, não existe esta necessidade.

6.7 Exercícios de Revisão

1. O que são operações *sobre bits*?
2. (a) Qual é o propósito do operador complemento-de-um? (b) Quais são os tipos de operandos que podem ser utilizados com este operador?
3. (a) Descreva os três operadores lógicos sobre bits. (b) Quais são os tipos de operandos que podem ser utilizados com esses operadores?
4. (a) O que é uma operação de mascaramento? (b) Qual é o propósito de cada operando numa operação deste tipo? (c) Como uma máscara é escolhida?
5. (a) Descreva utilizando diagramas uma operação de mascaramento na qual uma porção de um dado padrão de bits é copiado enquanto os bits restantes são todos igualados a zero. (b) Que operação sobre bits é utilizada para esta operação? (c) Como a máscara é selecionada?
6. (a) Descreva uma operação de mascaramento na qual uma porção de um dado padrão de bits é copiado enquanto os bits restantes são todos igualados a um. (b) Que operação sobre bits é utilizada para esta operação? (c) Como a máscara é selecionada? Compare este exercício com o **Exercício 5**.
7. (a) Descreva uma operação de mascaramento na qual uma porção de um dado padrão de bits é copiado enquanto os bits restantes são todos invertidos. (b) Que operação sobre bits é utilizada para esta operação? (c) Como a máscara é selecionada? Compare este exercício com os exercícios 5 e 6.
8. (a) Por que o operador complemento-de-um é algumas vezes usado em operações de mascaramento? (b) Em que condições seu uso é desejável?
9. (a) Como um determinado bit pode passar de 0 a 1, e vice-versa, alternadamente? (b) Que operador lógico bit-a-bit é utilizado para este propósito?
10. (a) Descreva as duas operações de deslocamento sobre bits. (b) Qual é o papel de cada operando? (c) Que requisitos os operandos devem satisfazer?
11. Por que nem todos os compiladores C tratam operações de deslocamento à direita da mesma maneira?
12. (a) Descreva os operadores de atribuição bit-a-bit. (b) Descreva cada operando numa operação de atribuição sobre bits.
13. (a) O que são **campos de bits**? (b) Como um campo de bits pode ser acessado?
14. Descreva as regras para definição de campos de bits.
15. Qual é o tipo de dado que deve ser associado a cada campo de bit?
16. O que acontece quando um campo de bit ultrapassa o limite de uma palavra de memória?
17. (a) Que interpretação é dada a um campo de bits anônimo? (b) Que interpretação é dada a um campo de bits de comprimento igual a zero?
18. Quais das seguintes chamadas de **printf()** produzem um resultado único e portátil?
 - (a) `printf("%x\n", ~0 >> 1);`
 - (b) `printf("%x\n", (unsigned) ~0 >> 1);`
 - (c) `printf("%x\n", (long) 1 << 32);`

6.8 Exercícios de Programação

EP6.1) Escreva uma função em C, denominada `DeslocamentoEsquerdoCircular()`, que recebe dois argumentos: o primeiro, `numero`, é um **unsigned long int** e o segundo, `deslocamento`, é um **unsigned char**. O objetivo da função é deslocar o primeiro argumento à esquerda um número de vezes determinado pelo segundo argumento, de modo que os bits de ordem superior sejam reintroduzidos como bits de ordem inferior. O protótipo da função é dado por:

```
long DeslocamentoEsquerdoCircular( unsigned long numero,  
                                   unsigned char deslocamento);
```

Por exemplo, se a representação binária de `numero` é dada por:

00010110 00111010 01110010 11100101

então, a chamada:

```
DeslocamentoEsquerdoCircular(numero, 5);
```

retornaria um valor **long int** cuja representação binária é dada por:

11000111 01001110 01011100 10100010

EP6.2) Escreva um programa em C que recebe lê um número em formato binário introduzido pelo teclado, converte-o para hexadecimal e imprime-o neste último formato.

EP6.3) Escreva uma função em C, chamada `Empacota()`, que recebe quatro caracteres como entrada e empacota-os em um **long int**.

Protótipo da Função: `long Empacota(char a, char b, char c, char d);`