CAPÍTULO 13

PROGRAMAÇÃO DE BAIXO NÍVEL EM C

13.1 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 de C serão apresentadas neste capítulo.

Manipulação de bits refere-se à realização de operações sobre bits considerados individualmente ou agrupamentos de bits menores do que um byte. Exemplos práticos de programas que incluem manipulação de bits são encontrados nas áreas de criptografia, computação gráfica, detecção e correção de erros e controladores de dispositivos (*drivers*).

Constantes inteiras utilizadas em operações de baixo nível são usualmente expressas em formato octal ou hexadecimal. Portanto, o assunto começa com uma breve revisão sobre sistemas de numeração utilizados em programação de baixo nível. A intenção aqui não é ensinar todos os fundamentos e práticas destes sistemas de numeração. Ao contrário, a intenção é recordar de modo abreviado aqueles requisitos que o aluno deve conhecer a fim de fazer bom proveito dos assuntos aqui discutidos.

Por uma questão de comodidade, muitos exemplos apresentados neste capítulo utilizam palavras de 16 bits que são raramente encontradas em computadores atuais. Contudo, o leitor não deverá apresentar nenhuma dificuldade para estender o raciocínio empregado nestes exemplos para

computadores com palavras de 32 ou 64 bits que são mais comuns hoje em dia.

13.2 Sistemas de Numeração para Manipulação de Bits

Como as operações apresentadas aqui envolvem manipulações de seqüências de bits, que são um tanto incômodas de serem escritas em formato **binário**, os formatos **octal** e **hexadecimal** são utilizados usualmente para representá-las. Portanto, antes de prosseguir, é importante que o aluno adquira a habilidade de transformar seqüências de bits nestes três formatos. A **Tabela 1**, apresentada a seguir, serve como referência auxiliar para essa tarefa.

FORMATO			
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 Numéricas

Para converter uma seqüência de bits do formato binário para hexadecimal, separe a seqüência dada em seqüências menores de quatro bits[1], completando, se for o caso, a primeira seqüência de quatro bits com zeros à esquerda. Por exemplo, a seqüência de bits `110101` pode ser separada em `0011` e `0101` (note que a primeira seqüência foi completada com zeros à esquerda). Após esta separação, 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 algum curso introdutório de computação. Se você não dominar esse assunto, poderá enfrentar dificuldades para entender o material apresentado no restante deste capítulo.

Em programação de baixo nível, que envolve manipulação de bits, o formato mais comumente utilizado é o formato hexadecimal. A **Tabela 2** apresenta algumas comparações que justificam porque este formato é preferido para manipulação de bits em detrimento dos demais.

[1] Dividir um número binário em seqüências de quatro bits não é um requisito fundamental. Isto é, este procedimento é adotado para apenas facilitar a conversão, pois qualquer dígito hexadecimal pode ser representado desta maneira.

FORMATO	VANTAGEM	DESVANTAGEM
Decimal	· Existe em C.	· Relativamente difícil de converter em binário. · Visualização das operações sobre bits não é trivial.
Binário	· Visualização imediata das operações.	· Com o crescimento das palavras dos computadores, tornou-se incômodo de usar. · Não existe em C.
Octal	· Facilidade de conversão para binário e vice-versa (separando-se o número binário em grupos de 3 bits) · Existe em C.	· Tamanhos comuns de palavras (16, 32, 64 bits) não são divisíveis por três. · Utiliza mais dígitos do que o formato hexadecimal. · Visualização das operações não é trivial.
Hexadecimal	· Facilidade de conversão para binário e vice-versa (separando-se o número binário em grupos de 4 bits) · Tamanhos comuns de palavras (16, 32, 64 bits) são divisíveis por 4. · Utiliza menos dígitos do que os demais formatos. · Existe em C.	· Visualização das operações não é trivial.

Tabela 2: Formatos de Constantes Inteiras Usados em Programação de Baixo Nível

Como constantes binárias não existem em C e podem ser confundidos com constantes em outros formatos, utiliza-se o subscrito 2 para identificar estas constantes sem ambigüidade. Constantes em formatos aceitos em C utilizam a notação da linguagem (v. **Seção 1.2.3**).

13.3 Operadores de Manipulação de Bits

Existem diversas situações em programação que 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. Eles podem ser divididos em três categorias:

- **Operadores lógicos sobre bits**
- **Operadores de deslocamento**
- **Operadores de atribuição sobre bits**

Estes operadores serão discutidos a seguir, mas, antes de prosseguir, é importante salientar que todos os operandos utilizados com eles devem ser inteiros. Operadores binários podem ter operandos de tipos diferentes (por exemplo, **char** e **unsigned long**), desde que sejam sempre inteiros. Quando tipos inteiros diferentes são misturados, o valor de um tipo de menor capacidade é convertido num valor do tipo de maior capacidade e o resultado é deste último tipo (v. **Seção 1.5**).

13.3.1 Operadores Lógicos sobre Bits

Existem um operador unário e dois operadores binários nesta categoria.

Operador de Complemento

O **operador de complemento**, representado pelo símbolo ~ (til), é um operador unário cujo efeito é o de inverter os bits de seu operando. Ou seja, 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, que deve ser de algum tipo inteiro. Para avaliar o resultado da aplicação do operador de complemento sobre um número em formato decimal, octal ou hexadecimal, siga a seguinte seqüência de passos:

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

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

1. `0xA52B` corresponde a 1010010100101011_2
2. `~0xA52B` é igual a 0101101011010100_2
3. $0101\ 1010\ 1101\ 0100_2$ corresponde a `0x5AD4`

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

O operador de complemento faz parte do mesmo grupo de precedência dos outros operadores unários de C e sua associatividade é da direita para a esquerda.

Operadores Lógicos Binários sobre Bits

Existem três operadores lógicos binários sobre bits, que são resumidamente apresentados na **Tabela 3**.

OPERADOR	SÍMBOLO
Conjunção sobre bits	&
Disjunção sobre bits	|
Disjunção exclusiva sobre bits	^

Tabela 3: Operadores Lógicos Binários sobre Bits

Cada um desses operadores requer dois operandos inteiros e as operações são executadas sobre cada par de bits correspondentes nos dois operando. Isto é, o primeiro bit do primeiro operando é avaliado com o primeiro bit do segundo operando, o segundo bit do primeiro operando é avaliado com o segundo bit do segundo operando, e assim por diante até que todos os bits sejam avaliados. O resultado destas avaliações para cada operador lógico sobre bits é apresentado na Tabela 4. Nesta tabela, `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 4: Resultados de Operações Lógicas Binárias sobre Bits

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

1. Transforme cada operando em binário;

2. Aplique o operador a cada par de bits correspondentes de acordo com a **Tabela 4;**

3. Escreva o resultado em formato 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 binário sobre bits tem sua própria precedência. Dentre eles, o operador **&** tem a maior precedência, depois vem o operador **^** e, finalmente, vem o operador **|**, que tem a menor precedência dos três. O operador **&** está numa classe de precedência imediatamente

abaixo daquela dos operadores de igualdade e desigualdade (**==** e **!=**). O operador **|** está numa classe de precedência imediatamente acima daquela do operador **&&**. Todos estes operadores têm associatividade da esquerda para a direita. Consulte o **Apêndice A** que apresenta um quadro completo de operadores da linguagem C.

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 1.3.6**. Na verdade, estes dois conjuntos de operadores possuem mais diferenças dos que semelhanças entre si e esta confusão freqüentemente resulta em erro de programação. Por exemplo, um erro bastante freqüente entre os iniciantes na linguagem C é usar o operador **&** ao invés do operador **&&**. A seguir, são enumeradas as principais diferenças entre os operadores lógicos, vistos na **Seção 1.3.6**, e os operadores lógico sobre bits apresentados aqui:

- Os operadores lógicos podem ser aplicados a operandos de quaisquer tipos primitivos de C (exceto **void**), 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, os bits que compõem os operandos são considerados individualmente. Por exemplo, os resultados das expressões apresentados na **Tabela 5** ilustram essas diferenças.

USANDO OPERADORES LÓGICOS	USANDO OPERADORES LÓGICOS SOBRE BITS
`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`

Tabela 5: Operadores Lógicos x Operadores Lógicos sobre Bits

• 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 uma divisão por zero. No entanto, se o operador **&&** for substituído por **&**, ocorrerá um erro em tempo de execução.

13.3.2 Operadores de Deslocamento

Existem dois **operadores de deslocamento** de bits em C, que são resumidamente apresentados na **Tabela 6**.

OPERADOR	SÍMBOLO
Deslocamento à esquerda	<<
Deslocamento à direita	>>

Tabela 6: Operadores de Deslocamento

Cada um desses operadores de deslocamento requer dois operandos inteiros: o primeiro operando representa uma seqüência de bits a ser deslocada, enquanto que o segundo representa o número de deslocamentos que cada bit do primeiro operando irá experimentar. O valor do segundo não deverá ser maior do que o número de bits utilizados para representar o primeiro.

A operação de deslocamento à esquerda faz com que os bits do primeiro operando sejam deslocados para a esquerda em um número de posições indicado pelo segundo. Os bits mais à esquerda do primeiro operando, em número igual ao valor do segundo, serão perdidos na

operação, enquanto que o bits mais à direita do primeiro, também em número igual ao valor do segundo, serão preenchidos com 0. Por exemplo, suponha que se deseje calcular o resultado da operação `0x6DB7 << 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 << 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 dele 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 >> 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. Logo, `0x6DB7 >> 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 zeros.

Se o primeiro 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.

Deslocar um número inteiro **unsigned** para a esquerda é equivalente a multiplicá-lo pela potência de dois do deslocamento. Isto é,

`x << y` 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 << 1` é equivalente a $5*2^1$, que é igual a 10, mas `1 << 31` é igual a -2^{31} e, portanto, a relação não é válida neste caso[2].

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

`x >> y` é equivalente a $x/2^y$

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

Quando o primeiro operando é **unsigned**, essas equivalências são seguramente verdadeiras, e as operações de deslocamento são mais eficientes do que as operações aritméticas equivalentes. Aliás, muitas outras operações aritméticas executadas utilizando operadores aritméticos convencionais podem ser efetuadas com uso de operadores sobre bits, e estas operações são também consideradas mais eficientes do que aquelas que utilizam operadores convencionais. O uso de operadores sobre bits para execução de operações aritméticas convencionais, no entanto, apresenta uma séria desvantagem: eles não resultam em expressões tão legíveis tanto quanto daquelas construídas utilizando operadores aritméticos convencionais. Portanto, o uso de operadores sobre bits neste

[2] Aqui estamos supondo que o tipo **int** ocupa 32 bits e, se você calcular 2^{31} obterá o mesmo resultado que 1 << 31, pois o maior valor que pode ser representado, neste caso, é $2^{31} - 1$.

contexto é recomendado apenas quando o desempenho do programa é realmente crítico.

É importante salientar que qualquer das operações de deslocamento apresentará um comportamento imprevisível e não-portável quando o segundo operando é negativo ou maior do que o tamanho do tipo do valor deslocado. Por exemplo, a expressão `1 << 88` resulta em `16777216`, quando o programa que a contém é copilado com o compilador Borland C++ 5.0 e resulta em 0 quando o programa é compilado com o compilador gcc 3.4.2.

13.3.3 Operadores de Atribuição sobre Bits

Como ocorre com os operadores aritméticos, a linguagem C dispõe de operadores que representam as operações sobre bits, vistas acima, combinadas com atribuição em operações únicas. Estas operações são representadas pelos **operadores de atribuição sobre bits** apresentados na **Tabela** 7.

OPERADOR	EXPRESSÃO	EXPRESSÃO EQUIVALENTE
&=	`a &= b`	`a = a & b`
|=	`a |= b`	`a = a | b`
^=	`a ^= b`	`a = a ^ b`
<<=	`a <<= b`	`a = a << b`
>>=	`a >>= b`	`a = a >> b`

Tabela 7: Operadores de Atribuição Bit a Bit

Os operadores de atribuição sobre bits têm a mesma precedência e a mesma associatividade que os outros operadores de atribuição apresentados na **Seção 1.6**. Além disso, o uso destes operadores é recomendado em situações semelhantes àquelas recomendadas para os operadores de atribuição aritmética (v. **Seção 1.6.3**).

13.4 Mascaramento

Mascaramento é uma operação na qual uma dada seqüência de bits é transformada numa outra 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**. Mais precisamente, uma máscara é um valor que, sendo usado em conjunto com uma dada operação sobre bits, é utilizado para extrair ou alterar o conteúdo de alguma variável.

Numa operação de mascaramento, a máscara e a operação lógica são escolhidas de modo que a operação resulte na seqüência desejada. Aplicações práticas de mascaramento incluem computação gráfica, processamento de imagens, endereçamento IP, uso de sinalizadores (***flags***) e muitos outros.

Existem vários tipos de operações de mascaramento. 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, conforme mostra a **Figura 1**.

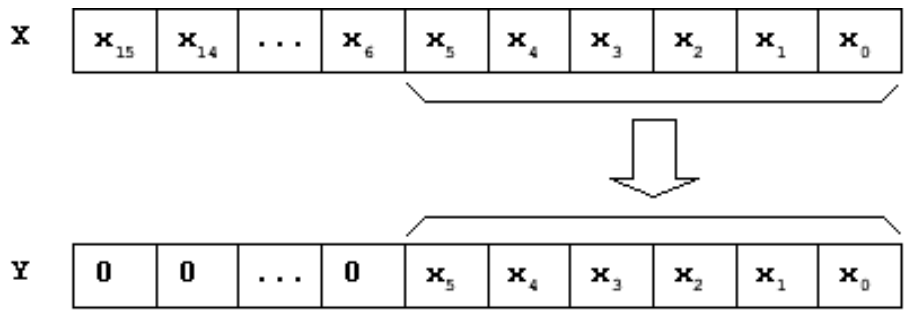


Figura 1: Operação de Mascaramento I

Suponha que `X` seja uma variável do tipo **int** e que se deseje extrair os seis bits mais à direita de `X` e atribuí-los a uma variável `Y`, como mostra o diagrama da **Figura 1**. 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? Para uma melhor visualização do problema, a operação de mascaramento é ilustrada diagramaticamente na Erro! A origem da referência não foi encontrada. apresentada a seguir[3].

[3] Aqui, é feita a suposição de que o tipo **int** ocupa 16 bits, mas isto é apenas comodidade e não constitui uma limitação.

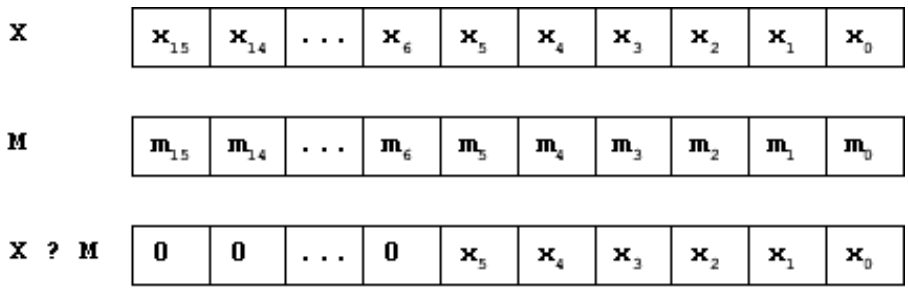


Figura 2: Operação de Mascaramento II

Nos diagramas das figuras acima, `X` é uma variável qualquer do tipo **int** e, conseqüentemente, os bits `x`$_i$ de `X` também são arbitrários. Os bits que compõem a máscara `M` devem ser escolhidos adequadamente, de modo a resultar na seqüência de bits dada por `X ? M`, onde `?` representa um operador sobre bits a ser convenientemente escolhido. Examinando-se a **Tabela 3** dos operadores sobre bits apresentada na **Seção 13.3.2**, pode-se concluir que esse operador deve ser **&** e que a máscara deve ser constituída por uma 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, o resultado da operação de mascaramento pode ser visualizados como apresentado na **Figura 3**.

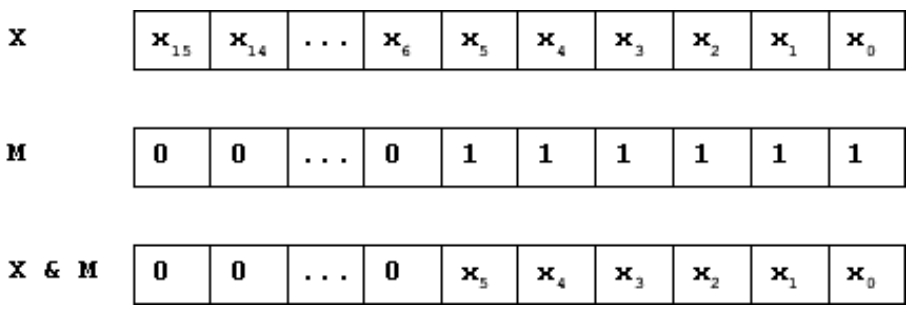


Figura 3: Operação de Mascaramento III

Portanto, a máscara da operação de mascaramento do exemplo acima é a constante com representação binária `0000 0000 0011 1111`, que corresponde, em formato hexadecimal, a `0x3F`.

É interessante notar 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, o valor da máscara continuaria sendo `0x3F`. Neste caso, a única diferença seria que a representação binária conteria mais zeros à esquerda, 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 mascaramento. Por exemplo, suponha que se deseje uma operação de mascaramento 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 anterior, vai verificar facilmente que o operador sobre bits a ser usado nesta nova operação continuará sendo **&**, 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 este resultado). Portanto, a máscara deste último exemplo é dependente de implementação, uma vez que o tamanho do tipo **int** depende do compilador utilizado. Felizmente, existe uma maneira de remover esta dependência: escrevendo a máscara em termos de seu complemento.

Quando se aplica o operador de complemento a um número, os zeros transformam-se em uns e os uns transformam-se em zeros. Sabe-se ainda que, para qualquer que seja o operando válido `Z`, a seguinte propriedade é válida:

~~Z é equivalente a Z

pois invertendo-se cada bit de `M` duas vezes, obtém-se o valor original de cada bit.

Portanto, se a máscara possui 1 em sua posição mais à esquerda, seu complemento terá 0 nesta mesma posição e, assim, tornar-se-á independente de implementação. Utilizando o operador de complemento, a operação de mascaramento do último exemplo não mais será representada por `X & M`, mas sim por `X & ~~M`. Ou seja, a operação de mascaramento apresentar-se-á como:

```
X & ~(complemento de M)
```

Como o complemento da máscara do último exemplo é dado por `0x3FF`, a operação de mascaramento pode ser apresentada como:

```
X & ~0x3FF
```

e o valor da máscara é independente de implementação.

Como em um outro exemplo de operação de mascaramento, suponha que se deseje copiar uma porção de uma seqüência de bits de uma dada palavra para uma nova palavra, com o restante da nova palavra sendo preenchida com uns. Aqui o operador **|** 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, se a porção de uns 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 mascaramento independente de implementação, deve-se utilizar o complemento da máscara como foi feito no último exemplo.

Como último exemplo de operação de mascaramento, uma seqüência de bits pode ser copiada para uma nova palavra, com os bits restantes sendo invertidos e copiados para a nova palavra. Esta operação

de mascaramento 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 a disjunção exclusiva sobre bits **^** 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 4**). Finalmente, o valor hexadecimal desta máscara é `0xFF` (confira) e este valor é independente de implementação. Se fosse desejado preservar os 8 bits mais à direita e inverter os 8 bits mais à esquerda, a operação de mascaramento deveria ser escrita em termos do operador de complemento para tornar-se independente de implementação (verifique isso).

13.5 Representação Interna de Valores Inteiros

Esta seção apresentará um exemplo prático de manipulação de bits em C. O exemplo consiste de uma função, denominada `RepresentacaoBinaria()`, que imprime, no meio de saída padrão, a seqüência de bits correspondente à representação binária de um valor do tipo **int** passado como parâmetro para a função.

```
#include <limits.h> /* CHAR_BIT é definida neste arquivo */

/****
 *
 * Função RepresentacaoBinaria()
 *
 * Descrição:Apresenta no meio de saída a representação binária
 *           de um valor do tipo int.
```

```
 *
 * Parâmetros:
 *      numero (entrada): valor inteiro cuja representação
 *                        binária será apresentada
 *
 * Retorno: Nada.
 *
 ****/

void   RepresentacaoBinaria(int  numero)
{
   int           i, bit, numeroDeBits;
   unsigned int  mascara; /* A máscara deve ser unsigned aqui */

       /* Cada byte contém um número de bits igual a CHAR_BIT */
    numeroDeBits  = CHAR_BIT*sizeof(int);

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

        /*  Imprime  cada  bit  a  partir  da  esquerda  do  número
(parâmetro) */
   for (i = 1; i <= numeroDeBits; i++) {
      bit = (numero & mascara) ? 1 : 0;
      printf("%d", bit);
      if (i % 4 == 0)   /* Separa bits em seqüências de quatro
para */
       printf("   "); /* melhor visualização da representação  */

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

A função `RepresentacaoBinaria()` merece alguns comentários:

- A constante **CHAR_BIT** é utilizada para calcular o número de bits utilizados pelo valor que será representado. Esta constante é definida no arquivo de cabeçalho `<limits.h>`[4].

[4] A constante CHAR_BIT existe porque a linguagem C não especifica quantos bits um byte deve possuir. Mas, mesmo que você não conheça nenhum computador cujo número de bits num byte seja diferente de 8, é melhor utilizar esta constante por questão de legibilidade.

• Diferentemente dos exemplos apresentados na **Seção 13.4**, a máscara utilizada aqui é representada por uma variável e não por uma constante. Em qualquer instante, esta máscara possui apenas um bit 1 e os demais bits são todos zeros. A posição do bit com valor 1 na máscara corresponde à posição do bit que se está testando se é 0 ou 1 no número recebido como parâmetro. Inicialmente, o bit mais à esquerda da máscara recebe o valor 1. Este bit é deslocado uma posição para a direita, utilizando o operador >>, a cada execução do corpo do laço **for**.

• No interior do laço, a expressão `numero & mascara` resulta num valor diferente de zero quando o bit do número a ser representado na posição correspondente ao bit com valor 1 na máscara for também igual a 1; caso contrário, este bit deve ser 0.

• A instrução **if** no interior do laço serve apenas para inserir alguns espaços em branco separando os bits da representação em agrupamentos de quatro bits para uma melhor visualização da representação.

A função `RepresentacaoBinaria()` apresentada acima poderia tornar-se mais geral de modo a imprimir a representação binária 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 valor que se deseja apresentar a representação binária e o tamanho do valor apontado pelo primeiro parâmetro. Em outras palavras, esta nova versão da função `RepresentacaoBinaria()` deveria ter como protótipo:

```
void  RepresentacaoBinaria2( const void *ptrValor,
                             size_t tamanhoDoValor )
```

A implementação da função `RepresentacaoBinaria2()` é deixada como exercício.

13.6 Campos de Bits

Em alguns programas, precisa-se trabalhar com tipos de dados que necessitam apenas de uns poucos bits de memória para ser 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 inúmeras 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 diferença sintática entre um campo de bits e um campo comum de uma estrutura é que, 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, com o qual o campo é declarado, na implementação utilizada.

Segundo o padrão ISO, o tipo de cada campo de bits deve ser **int** ou **_Bool** (C99), embora muitos compiladores permitam também que este tipo seja **short** ou **char**. Por exemplo, a estrutura `E` a seguir

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

é definida 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.

Observe 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 64 bits. Com o uso de campos de bits, apenas uma palavra (i.e., 16 bits) é suficiente para armazenar estes campos. Se há apenas uma ou umas poucas estruturas deste tipo num programa, a economia de memória é irrelevante, mas se houver um array de milhares de estruturas deste tipo, a economia poderá ser substancial.

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
- Uma função não pode retornar um campo de bits

De acordo com o padrão C99, um campo de bits pode ou não ultrapassar os limites de uma palavra em memória, dependendo da implementação. Em outras palavras, numa dada implementação, pode-se ter parte de um campo de bits numa palavra e outra parte em outra palavra, enquanto que em outra implementação o mesmo campo de bits faz parte de apenas uma 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, numa dada implementação, os 7 bits do campo d serão acomodados em outra palavra, e o bit restante na primeira palavra permanecerá desocupado. Neste caso, a segunda palavra terá 9 bits não utilizados. Mas, em outra implementação, o campo d poderá ter um bit armazenado numa palavra e os 6 bits restantes armazenados em outra palavra, que terá 10 bits não utilizados.

Campos de bits podem ser **anônimos**. Um campo de bits anônimo, obviamente, não pode ser acessado, 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. Isto é, o objetivo desse campo de bits anônimo é fazer o, assim chamado, **alinhamento** do campo c. Outra forma de controlar o alinhamento de campos de bits é o uso de um campo de bits anônimo de comprimento 0. Ou seja, o uso de um campo de bits anônimo de comprimento 0 força o campo que o segue a ser iniciado em outra palavra,

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;
```

Alinhamento de variáveis é discutido em maior profundidade no **Volume II**.

13.7 Aplicações Práticas

Esta seção apresenta três tipos de aplicações práticas de programação de baixo nível em C.

13.7.1 Sinalizadores

Suponha que um programa deva 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, verifica-se que existem 16 possibilidades de combinações de pressionamento destas teclas. Se o estado (pressionada ou não-pressionada) de cada tecla for representado como uma variável do tipo **unsigned char**, o que parece ser a escolha mais natural, o programa poderia ter o seguinte aspecto:

```
#define PRESSIONADA      1
#define NAO_PRESSIONADA  0
...
unsigned char direita,  /* Tecla direita    */
              esquerda, /* Tecla esquerda   */
              cima,     /* Tecla para cima  */
              baixo;    /* Tecla para baixo */
...
if (direita && esquerda && cima && baixo)
  ... /* Ação executada quando as quatro teclas estão pressionadas
*/
else if (direita && !esquerda && !cima && !baixo)
```

```
    ... /* Ação executada quando apenas a tecla direita é
pressionada */
  ...
   ... /* Outras possíveis combinações de teclas e respectivas
ações */
  ...
else if (!direita && !esquerda && !cima && !baixo)
  ... /* Ação executada quando nenhuma tecla está pressionada */
```

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

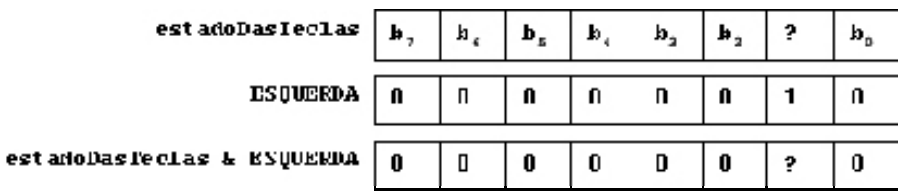
Em primeiro lugar, note que as variáveis `direita`, `esquerda`, `cima` e `baixo` são variáveis sinalizadoras (*flags*); isto é, variáveis que, em cada instante, assumem apenas um dentre dois possíveis valores. Este tipo de variável pode ser representado por um único bit e, no exemplo apresentado, as 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 identifiquem o pressionamento de cada tecla isoladamente e então utilizar operadores lógicos sobre bits para determinar o estado das teclas em conjunto. Para tornar a discussão mais tangível, suponha que se decida representar os quatro sinalizadores do programa nos bits 0, 1, 2 e 3 de uma variável do tipo **unsigned char** que representará o estado das quatro teclas. Então, poder-se-ía começar iniciar o novo programa como:

```
#define DIREITA  0x1 /* Tecla direita ocupa bit 0    */
#define ESQUERDA 0x2 /* Tecla esquerda ocupa bit 1   */
#define CIMA     0x4 /* Tecla para cima ocupa bit 2  */
#define BAIXO    0x8 /* Tecla para baixo ocupa bit 3 */
...
unsigned char estadoDasTeclas;
...
```

Neste último fragmento de programa, o valor da constante que define o pressionamento de uma dada tecla (`DIREITA`, `ESQUERDA` etc.) é

dado pela potência de dois da posição ocupada pelo respectivo sinalizador na variável `estadoDasTeclas`, que conterá todos os sinalizadores. Os valores são escolhidos assim porque estas constantes têm apenas um bit com valor 1, que é exatamente o bit correspondente àquele que a constante representa na variável que contém os sinalizadores. Por exemplo, o sinalizador da tecla direita ocupa o bit 0 da variável `estadoDasTeclas`; portanto, a constante `DIREITA` tem 1 em seu bit 0 e 0 nos demais bits. Ao invés de usar números mágicos (v. **Seção 6.6**) para representar potências de dois, é mais legível escrever essas constantes utilizando o operador de deslocamento <<, como:




```
#define DIREITA  (0x1 << 0) /* Tecla direita ocupa bit 0    */
#define ESQUERDA (0x1 << 1) /* Tecla esquerda ocupa bit 1   */
#define CIMA     (0x1 << 2) /* Tecla para cima ocupa bit 2  */
#define BAIXO    (0x1 << 3) /* Tecla para baixo ocupa bit 3 */
```

Prosseguindo com o exemplo, 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, a situação poderia ser descrita conforme esquematizado na **Figura 4**.

Figura 4: Testando se um Sinalizador Está Ligado

Portanto, conforme pode-se verificar, o resultado da operação `estadoDasTeclas & ESQUERDA` 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[5]. Assim, pode-se determinar se a tecla esquerda está pressionada utilizando um teste como no trecho de programa:

```
if (estadoDasTeclas & ESQUERDA)
...  /* Ação executada quando a tecla esquerda está pressionada
*/
else
...   /* Ação executada 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 respectivas teclas. Por exemplo, o fato de as teclas direita e esquerda estarem pressionadas pode ser expresso por:

```
DIREITA | ESQUERDA
```

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 DIREITA  (0x1 << 0) /* Tecla direita ocupa bit 0    */
#define ESQUERDA (0x1 << 1) /* Tecla esquerda ocupa bit 1   */
#define CIMA     (0x1 << 2) /* Tecla para cima ocupa bit 2  */
```

[5] O valor preciso do resultado não interessa no contexto atual.

```
#define BAIXO    (0x1 << 3) /* Tecla para baixo ocupa bit 3 */
...
unsigned char estadoDasTeclas = 0; /* Variável sinalizadora */
... /* Neste trecho é atribuído um valor à variável sinalizadora
*/
switch(estadoDasTeclas) {
   case DIREITA | ESQUERDA | CIMA | BAIXO:
    ... /* Ação executada quando as quatro teclas são pressionadas
*/
   case DIREITA | ESQUERDA:
        .../* Ação executada quando direita e esquerda são
pressionadas*/
   case DIREITA:
    ... /* Ação executada quando só a tecla direita é pressionada
*/
    ...
   default:
     ... /* Ação quando nenhuma tecla está pressionada */
}
```

Portanto, conforme demonstrado neste último trecho de programa, o aninhado de instruções **if** apresentado no início desta seção pode ser substituído por uma instrução **switch** que, conforme foi visto anteriormente (v. **Seção 1.7.4**), é mais legível do que aquele aninhado de instruções **if**. Além disso, no trecho de programa apresentado no início desta seção, são utilizadas quatro variáveis para conter os sinalizadores, enquanto que, aqui, apenas uma variável é suficiente para contê-los.

Agora, 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, sendo 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 sinalizadores, faça uma extrapolação. Suponha agora que você precise

trabalhar com trinta sinalizadores[6]. Neste caso, uma função que levasse em consideração todos os estados dos sinalizadores teria, no mínimo, trinta parâmetros, se a estratégia de uso de sinalizadores delineada aqui não fosse adotada. Adotando esta última estratégia, esses trinta sinalizadores ainda poderiam ser representados numa única variável inteira de 32 bits.

O uso de sinalizadores constitui aplicações práticas de operações de mascaramento, onde as constantes utilizadas são máscaras, usualmente denominadas **máscaras de bits**.

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

Ligando um Sinalizador

Pode-se **ligar** um sinalizador (i.e., tornar seu valor igual a 1) armazenado numa variável que contém um conjunto de sinalizadores, utilizando o operador **|=**. Por exemplo, considerando o caso apresentado no início desta seção, suponha que se deseje ligar o sinalizador que representa a tecla esquerda, independentemente do fato de a mesma já estar ligada ou não. Então, a operação seria esquematicamente representada como mostra a **Figura 5**.

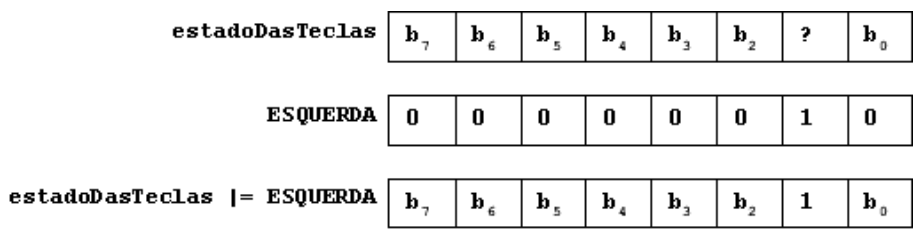


Figura 5: Ligando um Sinalizador

[6] Este número não é exageradamente irreal. Por exemplo, uma estrutura que representa uma janela de um sistema de janelas (como Microsoft Windows) pode possuir cerca de quarenta atributos, tais como estilo da borda, título, tipos de botões etc. Muitos destes atributos são representados por sinalizadores (por exemplo, se a janela possui botão "OK" ou não, se ela é modal ou não, se ela é flutuante ou não etc.).

Como mostra a **Figura 5**, ao final da operação `estadoDasTeclas |= ESQUERDA`, 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.

A função a seguir pode ser utilizada para ligar um sinalizador, representado pelo argumento `mascara`, armazenado num valor do tipo **unsigned int** contendo um conjunto de sinalizadores representado pelo argumento `sinalizadores`.

```
void LigaSinalizador(unsigned *sinalizadores, unsigned mascara)
{
   *sinalizadores |= mascara;
}
```

Desligando um Sinalizador

Pode-se **desligar** um sinalizador (i.e., tornar seu valor igual a 0) armazenado numa variável que contém um conjunto de sinalizadores, utilizando os operadores **~** e **&=**. Por exemplo, considerando o caso apresentado no início desta seção, suponha que se deseje desligar o sinalizador que representa a tecla esquerda, independentemente do fato de a mesma já estar desligada ou não. Então, a operação seria esquematicamente representada como na **Figura 6**.

	b_7	b_6	b_5	b_4	b_3	b_2	?	b_0
estadoDasTeclas								
ESQUERDA	0	0	0	0	0	0	1	0
estadoDasTeclas &= ~ESQUERDA	b_7	b_6	b_5	b_4	b_3	b_2	0	b_0

Figura 6: Desligando um Sinalizador

Como mostra a **Figura 6**, ao final da operação `estadoDasTeclas &= ~ESQUERDA`, o bit correspondente à tecla esquerda estará desligado, independentemente de seu valor inicial (fato indicado por "?" na **Figura 6**). Note ainda que, novamente, os demais bits da variável `estadoDasTeclas` não são modificados pela operação.

A função a seguir pode ser utilizada para desligar um sinalizador, representado pelo argumento `mascara`, armazenada num valor do tipo **unsigned int** contendo um conjunto de sinalizadores representado pelo argumento `sinalizadores`.

```
void DesligaSinalizador(unsigned *sinalizadores, unsigned
mascara)
{
   *sinalizadores &= ~mascara;
}
```

Invertendo um Sinalizador

Pode-se **inverter** um sinalizador (i.e., trocar seu valor) armazenado numa variável que contém um conjunto de sinalizadores, utilizando o operador **^=**. Por exemplo, considerando o caso apresentado no início desta seção, suponha que se deseje inverter o sinalizador que representa a tecla esquerda (independentemente de seu valor corrente). Esta operação seria esquematicamente representada como mostra a **Figura 7**.

Figura 7: Invertendo um Sinalizador

Na ilustração da **Figura 7**, ? representa 0 ou 1. Conseqüentemente, ~? irá representar 1 ou 0, respectivamente. Para 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 será invertido. Note que os demais bits da variável estadoDasTeclas não são modificados pela operação.

A função a seguir pode ser utilizada para inverter um sinalizador, representada pelo argumento mascara, armazenado num valor do tipo **unsigned int** contendo um conjunto de sinalizadores representado pelo argumento sinalizadores.

```
void InverteSinalizador(unsigned *sinalizadores, unsigned
mascara)
{
   *sinalizadores ^= mascara;
}
```

Testando um Sinalizador

A última operação comum a sinalizadores armazenados numa variável é **testar** se um sinalizador está ligado 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 um sinalizador, representado pelo argumento mascara, armazenada num valor do tipo **unsigned int** contendo um conjunto de sinalizadores, representado pelo argumento sinalizadores, está ligado ou não.

```
unsigned TestaSinalizador( unsigned sinalizadores, unsigned
mascara )
{
   return sinalizadores & mascara;
}
```

Um Exemplo Completo

Para concluir o estudo sobre sinalizadores, será apresentado a seguir um programa que demonstra o uso dos conceitos e operações vistos nesta seção. Este programa dispensa comentários adicionais, visto que ele simplesmente reúne operações com sinalizadores fartamente discutidas acima.

```
#include <stdio.h>

#define DIREITA  (0x1 << 0) /* Tecla direita ocupa bit 0    */
#define ESQUERDA (0x1 << 1) /* Tecla esquerda ocupa bit 1   */
#define CIMA     (0x1 << 2) /* Tecla para cima ocupa bit 2  */
#define BAIXO    (0x1 << 3) /* Tecla para baixo ocupa bit 3 */

void LigaSinalizador(unsigned *sinalizadores, unsigned mascara)
{
   *sinalizadores |= mascara;
}

void   DesligaSinalizador(unsigned   *sinalizadores,   unsigned
mascara)
{
   *sinalizadores &= ~mascara;
}

void   InverteSinalizador(unsigned   *sinalizadores,   unsigned
mascara)
{
   *sinalizadores ^= mascara;
}

unsigned  TestaSinalizador(  unsigned  sinalizadores,  unsigned
mascara )
{
   return sinalizadores & mascara;
}

void ImprimeStatusTeclas(unsigned teclas)
{
   switch(teclas) {
      case DIREITA | ESQUERDA | CIMA | BAIXO:
         printf("Todas as teclas estao ligadas\n");
         break;
      case DIREITA | ESQUERDA | CIMA:
```

```
        printf("As teclas direita, esquerda e cima estao ligadas\
n");
            break;
         case DIREITA | ESQUERDA | BAIXO:
            printf("As teclas direita, esquerda e baixo "
                   "estao ligadas\n");
            break;
         case DIREITA | CIMA | BAIXO:
            printf("As teclas direita, cima e baixo estao ligadas\
n");
            break;
         case ESQUERDA | CIMA | BAIXO:
           printf("As teclas esquerda, cima e baixo estao ligadas\
n");
            break;
         case DIREITA | ESQUERDA:
             printf("As teclas direita e esquerda estao ligadas\
n");
            break;
         case DIREITA | CIMA:
            printf("As teclas direita e cima estao ligadas\n");
            break;
         case DIREITA | BAIXO:
            printf("As teclas direita e baixo estao ligadas\n");
            break;
         case ESQUERDA | CIMA:
            printf("As teclas esquerda e cima estao ligadas\n");
            break;
         case ESQUERDA | BAIXO:
            printf("As teclas esquerda e baixo estao ligadas\n");
            break;
         case CIMA | BAIXO:
            printf("As teclas cima e baixo estao ligadas\n");
            break;
         case DIREITA:
            printf("A tecla direita esta' ligada\n");
            break;
         case ESQUERDA:
            printf("A tecla esquerda esta' ligada\n");
            break;
         case CIMA:
            printf("A tecla cima esta' ligada\n");
            break;
         case BAIXO:
            printf("A tecla baixo esta' ligada\n");
            break;
         default:
```

```
            printf("Nenhuma tecla esta' ligada\n");
    }
}

int main(void)
{
    unsigned teclas = 0;

    LigaSinalizador(&teclas, ESQUERDA);
    ImprimeStatusTeclas(teclas);

    DesligaSinalizador(&teclas, ESQUERDA);
    ImprimeStatusTeclas(teclas);

          /* Inverte ao mesmo tempo os bits correspondentes às
teclas */
         /* CIMA e BAIXO; como eles estão zerados, serão ligados
*/
    InverteSinalizador(&teclas, CIMA | BAIXO);
     ImprimeStatusTeclas(teclas);

    if (TestaSinalizador(teclas, CIMA))
         printf("TestaSinalizador(): a tecla cima esta' ligada\
n");
    else
       printf("TestaSinalizador(): a tecla cima NAO esta' ligada\
n");

        /* Neste ponto, as teclas CIMA e BAIXO estão */
        /* ligadas; a seguir, apenas a tecla CIMA é  */
        /* desligada; a tecla BAIXO continua ligada  */
    DesligaSinalizador(&teclas, CIMA);
    ImprimeStatusTeclas(teclas);

        /* A instrução a seguir desliga todas as teclas. */
        /* Evidentemente, isto poderia ser feito de modo */
        /* mais simples com atribuição: teclas = 0;      */
     DesligaSinalizador(&teclas, DIREITA | ESQUERDA | CIMA |
BAIXO);
    ImprimeStatusTeclas(teclas);

    return 0;
}
```

Quando o programa acima é executado, ele imprime como resultado:

```
A tecla esquerda esta' ligada
Nenhuma tecla esta' ligada
As teclas cima e baixo estao ligadas
TestaSinalizador(): a tecla cima esta' ligada
A tecla baixo esta' ligada
Nenhuma tecla esta' ligada
```

13.7.2 Criptografia XOR

O operador **^**, também denominado operador **xor**[7], tem propriedades interessantes:

1. `x ^ x` resulta em zero, para qualquer que seja o operando válido `x`.
2. `(x ^ y) ^ y` resulta sempre em `x`, para quaisquer que sejam os operandos válidos `x` e `y`. Isto significa dizer 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 ^ y` é o mesmo que `y ^ x`, para quaisquer que sejam os operandos válidos `x` e `y`. Ou seja, o operador **^** é comutativo.

Estas propriedades são fáceis de demonstrar formalmente num curso sobre Álgebras de Boole, mas o que interessa aqui são algumas conseqüências práticas destas propriedades.

Uma conseqüência da primeira propriedade é que se pode atribuir 0 a uma variável inteira atribuindo-lhe o resultado da operação *xor* da variável consigo mesma. Ou seja, utilizando o operador de atribuição **^=**, pode-se atribuir zero a uma variável `x` por meio da instrução[8]:

```
x ^= x;
```

---

[7] A denominação *xor* vem da expressão *exclusive or* em inglês; i.e., ou exclusivo, em português.

[8] Isto não significa, entretanto, que esta forma de atribuição seja 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. Este fato é apresentado aqui mais por uma questão de curiosidade do que por uma questão prática.

Uma aplicação da segunda propriedade de *xor* apresentada acima é que ela permite trocar os valores de duas variáveis inteiras sem a utilização de nenhuma 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 ^ y`
2. Faça `y` receber `x ^ y`
3. Faça `x` receber `x ^ 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 ^ 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 ^ y) ^ y`, onde `x` representa o valor original de `x` e, de acordo com a segunda propriedade de *xor* vista acima, `y` estará recebendo `x` (valor original).

- No terceiro passo, `x` recebe o valor da expressão `x ^ y`, mas, como no passo anterior, novamente `x` aqui representa o valor `x ^ 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 ^ y` no passo 3 significa `(x ^ y) ^ 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 **^=**, o algoritmo de troca descrito acima pode ser implementado como na seguinte função:

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

Outra aplicação prática das propriedades do operador *xor* é a implementação de um método para cifrar arquivos denominado **criptografia *xor***.

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 bem como introdução didática ao tema.

Existem várias variantes de criptografia *xor*, 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 denominado **chave criptográfica**. Então, o programa executa uma operação *xor* sobre cada byte no arquivo com a chave criptográfica. Em conseqüência 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>

/****
 *
 * Função Criptografa():
 *
 * Descrição: Faz criptografia xor de um arquivo.
 *
```

```
 * Parâmetros:
  *          arquivo (entrada): nome do arquivo que será
criptografado
 *      chave (entrada): caractere que representa a
 *                       chave criptográfica
 *
 * 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 escrita 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, executa xor do byte lido
*/
       /* com a chave e grava o resultado 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 retornar 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 1;
   }

   return 0;
}
```

A função `Criptografa()` 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 binário. Embora talvez o maior interesse seja a criptografia de arquivos de texto, este 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.

A função utiliza ainda um arquivo temporário para armazenar transitoriamente o resultado da criptografia. Ele é criado e automaticamente aberto no modo `"wb+"` por meio da instrução:

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

Este arquivo é automaticamente destruído quando é fechado utilizando a função **fclose()** ou quando o programa é encerrado (v. **Seção 12.10.6**).

A criptografia propriamente dita é realizada por meio 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 no arquivo original Antes disso, porém, é necessário retornar o apontador de posição ao início de cada arquivo. Isto é feito por meio de chamadas da função **rewind()**. Outros detalhes de funcionamento da função `Criptografa()` e da função **main()** que a utiliza são apresentados em forma de 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 tentativas, que corresponde ao número de valores possíveis para o tipo **char**, que é o tipo do valor utilizado como chave criptográfica. 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).

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. Uma nova versão da função de criptografia *xor*, denominada `Criptografa2()`, que implementa esta idéia é apresentada a seguir.

```
/****
 *
 * Função Criptografa2():
 *
 * Descrição: Faz criptografia xor de um arquivo.
 *
 * Parâmetros:
  *            arquivo  (entrada):  nome  do  arquivo  que  será
criptografado
  *          chave  (entrada):  string  que  representa  a  chave
criptográfica
 *
  * 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 array sequenciaBytes        */
      /* será utilizado para esta finalidade.              */
   tamanhoChave = strlen(chave);
   sequenciaBytes = malloc(tamanhoChave);

      /* Se não foi possível alocar espaço para o array */
      /* sequenciaBytes, considera encerrada a operação */
      /* e retorna indicando a falha. Mas, existem      */
      /* alternativas que não foram consideradas aqui.  */
   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);

       if (ferror(ptrTemp)) /* Se ocorreu algum erro de escrita
*/
         break;            /* no arquivo temporário, sai do laço
*/
   } while ( nBytesLidos == tamanhoChave );
```

```
        /* Verifica se houve erro de leitura/escrita. */
        /* Se for o caso, fecha arquivos e retorna.   */
   if (ferror(ptrArquivo) || ferror(ptrTemp)) {
      fclose(ptrArquivo);
      fclose(ptrTemp);
      return 0;
   }

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

        /* Copia conteúdo do arquivo temporário para 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);   /* Sistema remove arquivo temporário  */

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

   return 1;
}
```

A principal alteração apresentada nesta última versão da função de criptografia *xor* com relação à versão anterior 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 ou quando ocorre algum erro (v. **Seção 12.8.3**).

Esta versão da função que executa criptografia *xor* testa se ocorre algum erro de leitura ou escrita e aborta a operação quando isso ocorre. Este procedimento também deveria ter sido adotado pela primeira versão da função que executa a criptografia, mas ela não foi incluída naquela versão para torná-la o mais simples possível. As partes restantes desta função são semelhantes àquelas da primeira versão apresentada anteriormente e são auto-explicativas.

13.7.3 Endereçamento IP

Um **endereço IP** (*Internet Protocol*) identifica de maneira única um nó ou servidor de uma rede IP. Tal endereço 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 parte identifica a rede e 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 que se distinguem pelo valor do primeiro octeto, conforme mostrado na **Tabela 8**.

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

Tabela 8: Endereçamento IP: Classes de Endereços

Sabendo a que classe pertence um dado endereço, as partes pertencentes à rede (`R`) e ao nó (`N`) são assim determinadas[9]:

- Classe A: `RRRRRRRR.NNNNNNNN.NNNNNNNN.NNNNNNNN`
- Classe B: `RRRRRRRR.RRRRRRRR.NNNNNNNN.NNNNNNNN`
- Classe 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 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 nó do endereço. As máscaras padrão das classes A, B e C são apresentadas, em formatos binário e decimal, na **Tabela 9** 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

Tabela 9: Endereçamento IP: Máscaras Padrão das Classes A, B e C

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:

[9] Apenas as classe A, B e C são de interesse daqui em diante.

```
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-rede[6]. 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.

13.8 Exercícios de Revisão

1. O que é manipulação de bits?

2. (a) Qual é o propósito do operador de complemento? (b) Quais são os tipos de operandos que podem ser utilizados com este operador?

3. A macro a seguir pode ser utilizada para trocar os valores de dois números inteiros:

```
#define TROCA(a, b) do {a ^= b; b ^= a; a ^= b;}
                    while(0)
```

(a) Por que os argumentos desta macro não precisam estar envolvidos entre parênteses conforme recomendado na **Seção 5.3**?

(b) Compare esta macro com a função `Troca()` apresentado na **Seção 3.3.3**.

4. (a) Descreva os três operadores lógicos binários sobre bits. (b) Quais são os tipos de operandos que podem ser utilizados com esses operadores?

5. Que valor será atribuído a `x` após a avaliação da segunda expressão a seguir quando o tipo **int** ocupa (a) 16 bits e (b) 32 bits. (c) Esta expressão é portável? (d) Se for o caso, como tornar esta expressão portável?

```
int x = 25;
x &= 0xFFF8;
```

6. A expressão a seguir realiza uma operação bem comum em programação, mas de modo mais eficiente do que por meio de métodos convencionais. Qual é esta operação?

```
z = y + ((x - y) & -(x < y));
```

7. Suponha que `x`, `y` e `z` sejam variáveis do tipo **unsigned char** e que `N` seja uma constante tal que `1 ≤ N ≤ CHAR_BIT`, onde **CHAR_BIT** é

número de bits utilizados para representar um valor do tipo **char**. Qual o efeito colateral sobre `x`, `y` e `z` resultante das seguintes operações:

```
(a) x |= (1 << n);

(b) y &= ~(1 << n);

(c) c ^= (1 << n);
```

8. (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?

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

10. Escreva uma operação de mascaramento, 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.

11. (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?

12. (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 anterior.

13. (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 dois últimos exercícios.

14. (a) Por que o operador complemento de um é algumas vezes usado em operações de mascaramento? (b) Em que condições seu uso é desejável?

15. (a) Como um determinado bit pode passar de 0 a 1 e vice-versa, alternadamente? (b) Que operador lógico sobre bits é utilizado para este propósito?

16. (a) Descreva as duas operações de deslocamento sobre bits. (b) Qual é o papel de cada operando? (c) Que requisitos os operandos devem satisfazer?

17. Compiladores de C nem sempre tratam operações de deslocamento à direita da mesma maneira. Em que situações as operações de deslocamento à direita não são portáveis.

18. (a) Descreva os operadores de atribuição sobre bits. (b) Descreva cada operando numa operação de atribuição sobre bits.

19. (a) O que são campos de bits? (b) Descreva as regras para definição de campos de bits. (c) Como um campo de bits pode ser acessado?

20. O que é e para que serve um campo de bits anônimo?

21. Que tipos de dados podem ser associado a um campo de bits?

22. Que interpretação é dada a um campo de bits de comprimento igual a zero?

23. Quais das seguintes chamadas de **printf()** produzem um resultado único e portável?

```
(a) printf("%x\n", ~0 >> 1);
(b) printf("%x\n", (unsigned) ~0 >> 1);
(c) printf("%x\n", (long) 1 << 32);
```

24. O que as funções `Misterio1()` e `Misterio2()`, definidas a seguir, calculam?

```
unsigned Misterio1(unsigned a, unsigned b)
{
   unsigned x = 0,
            y = 0,
            z = ~0;

   for (z = ~0; z; z >>= 1) {
      y <<= 1;
      y |= (a^b^x) & 1;
      x = ((a | b) & x | a & b) & 1;
      a >>= 1;
      b >>= 1;
   }

   for (z = ~0, x = ~z; z; z >>= 1) {
      x <<= 1;
      x |= y & 1;
      y >>= 1;
   }

   return x;
}

unsigned Misterio2(unsigned a, unsigned b)
{
   unsigned resultado;

   for (resultado = 0; a; b <<= 1, a >>= 1)
      if (a&1)
         resultado = Misterio1(resultado, b);

   return resultado;
}
```

25. (a) Qual seria o nome mais apropriado para a função `F()` a seguir? (b) Qual seria o nome mais apropriado para a variável local `retorno` desta função?

```
unsigned F(int x)
{
   unsigned retorno = 0;

   while (x) {
      if (x & 0x1)
         retorno++;
```

```
      x >>= 1;
   }

   return retorno;
}
```

26. Demonstre que as seguintes propriedades do operador *xor* são válidas:

(a) Para qualquer inteiro `x`, `x ^ x = 0`

(b) Para quaisquer inteiros `x` e `y`, `(x ^ y) ^ y = x`

(c) Para quaisquer inteiros `x` e `y`, `x ^ y = y ^ x`

27. Por que o programa a seguir entra em laço infinito?

```
#include <stdio.h>

int main()
{
   short int   i;
   signed char c;

   for (i = 0x80; i != 0; i = i >> 1) {
      printf("i = %x (%d)\n", i, i);
   }

   for (c = 0x80; c != 0; c = c >> 1) {
      printf("c =  %x (%d)\n", c, c);
   }

   return 0;
}
```

13.9 Exercícios de Programação

EP13.1) Escreva uma função em C, denominada `DeslocamentoEsquerdoCircular()`, que recebe dois argumentos: o primeiro, chamado `numero`, é do tipo **unsigned long int** e o segundo, chamado `deslocamento`, é do tipo **unsigned int**. 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 int
deslocamento)
```

Por exemplo, se a representação binária do parâmetro `numero` fosse dada por:

```
00010110    00111010    01110010          11100101
```

então, a chamada:

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

retornaria um valor **long int** cuja representação binária seria:

```
11000111    01001110    01011100          10100010
```

EP13.2) Escreva um programa em C que lê um número em formato binário introduzido pelo teclado, converte-o para hexadecimal e imprime-o neste último formato.

EP13.3) Escreva uma função, chamada `Empacota()`, que recebe quatro caracteres como entrada e *empacota-os* em um **long int**. Utilize o seguinte protótipo para esta função:

```
long  Empacota(char a, char b, char c, char d)
```

EP13.4) Escreva um programa que recebe um valor do tipo **unsigned char** representando um conjunto de sinalizadores e oferece opções que permitem ao usuário executar o seguinte conjunto de operações:

- Introduzir um novo valor
- Ligar um bit do valor
- Desligar um bit do valor
- Inverter um bit do valor
- Testar um bit do valor
- Sair do programa

O programa deverá apresentar a representação binária do valor introduzido pelo usuário após a execução de cada operação. [**Sugestão**: Adapte a função `RepresentacaoBinaria()` apresentada na **Seção 13.5** para esta finalidade.] Além disso, a opção introduzida pelo usuário não deve requerer que o mesmo pressione `[ENTER]` ou `[RETURN]`. [**Sugestão**: utilize a função `getche()` (ou equivalente) do módulo `conio` (ou equivalente), mas lembre-se que esta função não faz parte da biblioteca padrão e, portanto, você deve confiná-la em outra função – v. **Seção 12.10.9**.]

Exemplo de interação com o programa (**negrito** significa digitação do usuário):

```
Pressione 1 para introduzir um valor
Pressione 2 para ligar um bit
Pressione 3 para desligar um bit
Pressione 4 para inverter um bit
Pressione 5 para testar um bit
Pressione 6 para sair do programa

Opcao: 1
Introduza um valor representando um conjunto de flags (0-
255): 12
Valor corrente:  00001100 = 12

Opcao: 2
Qual e' o bit  (0-7) a ser ligado?  5
Valor corrente:  00101100 = 44

Opcao: 3
Qual e' o bit  (0-7) a ser desligado?  1
O bit 1 ja' esta' desligado
Valor corrente:  00101100 = 44
```

```
Opcao: 4
Qual e' o bit  (0-7) a ser invertido?  1
Valor corrente:  00101110 = 46

Opcao: 5
Qual e' o bit  (0-7) a ser testado?  3
O bit 3 esta' ligado
Valor corrente:  00101110 = 46

Opcao: 6
Bye, bye.
```

EP13.5) Escreva que um programa que recebe um nome de um arquivo de texto como entrada e imprime o conteúdo deste arquivo criptografado/não-criptografado em outro arquivo utilizando criptografia *xor* de acordo com uma chave (caractere) provida pelo usuário.

EP13.6) Generalize a função `RepresentacaoBinaria()` apresentada na **Seção 13.5**, de modo a nova versão possa imprimir a representação binária de um parâmetro de qualquer tipo. Esta nova versão da função `RepresentacaoBinaria()` deve ter como protótipo:

```
void  RepresentacaoBinaria2( const void *ptrValor,
                             size_t tamanhoDoValor )
```

EP13.7) Escreva um programa capaz de descobrir a chave criptográfica de um arquivo criptografado conforme descrito na **Seção 13.7.2** sabendo que o arquivo original é um arquivo de texto. (**Sugestão**: Seu programa pode utilizar um laço de repetição no qual 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 decifrados ao usuário e perguntaria a ele 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.)

EP13.8)

(a) Escreva uma função, cujo protótipo é:

```
AlteraBits(unsigned char *x, unsigned p, unsigned n, y)
```

que altera o os `n` bits do parâmetro `x` que começam na posição `p` recebendo os `n` bits mais à direita do parâmetro, deixando os demais bits de `x` inalterados. Por exemplo, se `x = 10101010`$_2$ (`170`$_{10}$), `y = 10100111`$_2$ (`167`$_{10}$), `n = 3` e `p = 6` os 3 bits mais à direita de `y` (`111`) são colocados na posição 3 de `x` resultando em `10111010`$_2$.

(b) Escreva um programa que recebe como entrada os dados necessários para chamar a função `AlteraBits()` em formato decimal, chama esta função e apresenta o resultado em formato binário.

Considere o seguinte exemplo de interação:

```
Valor a ser alterado (x): 123
Origem dos bits (y): 123
Numero de bits a ser alterados: 3
Posicao (p): 2

x = 10101010 (binário)
y = 10100111 (binário)
n = 3
p = 6

Resultado: x = 10111010 (binário)
```

EP13.9) (a) Escreva uma função, denominada `InverteBits()`, que inverte os bits de um parâmetro do tipo **unsigned int**. (b) Escreva um programa que recebe como entrada um valor do tipo **unsigned int**, converte-o utilizando a função `InverteBits()` e apresenta o valor original recebido como entrada e o valor alterado pela função `InverteBits()` em formato binário.

EP13.10) Uma forma relativamente simples (mas insegura) de criptografar um arquivo de texto é aplicando o algoritmo conhecido

como **Rot13** sobre cada caractere do arquivo. Este algoritmo pode ser implementado da seguinte maneira:

```
char Rot13(char c)
{
   if('A' <= c && c <= 'Z')
      return(((c - 'A' + 13) % 26) + 'A');
   else if('a' <= c && c <= 'z')
      return(((c -' a' + 13) % 26) + 'a');
   else
      return c;
}
```

(a) Escreva um programa em C que lê um arquivo de texto especificado pelo usuário em linha de comando e criptografa-o utilizando a função Rot13().

(b) Examine detalhadamente a função Rot13() e tente descobrir por que este algoritmo é denominado Rot13.

(c) Mostre que o mesmo algoritmo utilizado para criptografar um arquivo também serve para decifrá-lo. (Sugestão: Existem inúmeros artigos relacionados ao algoritmo Rot13 na internet. Encontre um destes arquivos providos por uma fonte confiável, estude-o e você descobrirá a resposta.)