

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 flags¹. Neste caso, uma função que levasse em 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

¹ 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 representados por flags (por exemplo, se a janela possui botão OK ou não, se ela é modal ou não, etc.).

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.

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 \wedge , também denominado operador xor², tem propriedades interessantes:

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 instrução³:

```
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**⁴. Existem várias variantes deste tipo de criptografia,

² A denominação *xor* vem do Inglês *exclusive or*; i.e., ou-exclusivo.

³ 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.

⁴ 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

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 */
}
```

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.

```

    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 binário⁵. 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()`⁶.

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 tentativas⁷, 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

⁵ 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.

⁶ O arquivo temporário também é destruído automaticamente quando o programa é encerrado.

⁷ 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 determinadas⁸:

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

⁸ Apenas as classe A, B e C são de interesse daqui em diante.

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-rede⁹. 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.

⁹ Se os nós estiverem em redes diferentes, o pacote enviado precisará ser roteado para outra rede; caso contrário, não existe esta necessidade.