MAC115 – Introdução à Computação para Ciências Exatas e Tecnologia
Instituto de Física – Novembro de 2012

Quarto Exercício-Programa (EP4)
Data de entrega: 02/12/2012

1 Compactador LZW

Em Ciência da Computação, a compressão de dados é uma técnica que tem como objetivo codificar a informação usando menos bits que sua representação original. A compressão é útil por auxiliar a reduzir o consumo de recursos computacionais tais como espaço de armazenamento de dados ou banda de transmissão. Exemplos bastante conhecidos de formatos de dados compactados são o *zip*, o *rar* e o *tar*.

Um algoritmo de compressão sem perdas reduz a quantidade de bits por meio da identificação e eliminação da redundância estatística dos dados; nesse tipo de compressão, nenhuma informação é perdida. Por exemplo, uma imagem pode possuir áreas constituídas por longas sequências de pixels com uma mesma cor; nesse caso, em lugar de uma codificação do tipo "pixel vermelho, pixel vermelho, ...", os dados poderiam ser codificados como "279 pixels vermelhos".

O objetivo deste exercício programa é implementar um algoritmo de compressão sem perdas – o LZW (Lempel-Ziv-Welch). O LZW é derivado do algoritmo LZ78, baseado na localização e no registro das padronagens de uma estrutura. Foi desenvolvido e patenteado em 1984 por Terry Welch. O codificador LZW reduz costuma reduzir arquivos de imagens gráficas a 1/3 ou 1/4 de seu tamanho original. Imagens com padronagem bem definidas – com grandes blocos de cor contínua – podem ser reduzidas a 1/10 do tamanho original do arquivo[1]. Apesar da sua eficácia para a compactação de arquivos de imagens, neste EP o LZW será usado na compressão de arquivos de texto.

1.1 Compactação de arquivos de texto

Um arquivo de texto é uma sequência de caracteres. Cada caractere possui um código da tabela ASCII associado a ele. Um código ASCII é um número de 8 bits (1 byte), na faixa entre -128 e 127. Logo, podemos pensar em um arquivo de texto como sendo uma sequência de números entre -128 e 127.

– E se tivéssemos códigos que, em lugar de representar um único caractere, representassem uma sequência de 1 ou mais caracteres? R: Conseguiríamos codificar um texto com uma sequência de números menor do que a que é necessária quando usamos somente os códigos da tabela ASCII.

O algoritmo de compactação LZW (assim como outros algoritmos de compactação) explora essa ideia para compactar arquivos de texto. Ele identifica as sequências de 2 ou mais caracteres que aparecem no arquivo de entrada e associa a essas sequências um novo código. A primeira sequência identificada recebe o código 128 (que é o número seguinte ao último código usado na tabela ASCII); os códigos seguintes são atribuídos às novas sequências à medida em que elas são identificadas. Os pares sequência–código identificados pelo algoritmo são mantidos em um dicionário (= tabela), que aumenta à medida em que o arquivo de entrada é lido caractere por caractere. As entradas desse dicionário mais a própria tabela ASCII são então usadas para codificar os dados do arquivo de entrada, resultando em um arquivo compactado.

Na descompactação, o dicionário também é criado de forma dinâmica, à medida em que os dados do arquivo compactado são lidos. Esse dicionário mais a própria tabela ASCII são usados para decodificar o arquivo

[1] Informações extraídas de: `http://pt.wikipedia.org/wiki/LZW`.

compactado e gerar o arquivo de texto original.

Os detalhes envolvidos na compactação e descompactação de arquivos de texto usando o LZW são apresentados nas seções a seguir.

2 Compactação

O Algoritmo 1 descreve os passos envolvidos na compactação de um arquivo de texto usando o método de compactação LZW. O algoritmo recebe como entrada um arquivo de texto contendo os dados a serem compactados e gera como saída um arquivo binário contendo uma sequência de códigos numéricos que representam os dados do arquivo de entrada de forma mais compacta.

Algoritmo 1 Compactação

```
1: crie um dicionário vazio
2: abra o arquivo de texto de entrada para leitura
3: abra o arquivo binário de saída para escrita
4: entrada ← " " {a string entrada começa vazia}
5: novo_codigo ← 128 {primeiro código maior que os da tabela ASCII}
6: leia um caractere do arquivo de entrada e armazene-o na variável c
7: enquanto não é o fim do arquivo de entrada faça
8:    se o comprimento de entrada+c (ou seja, a string formada por entrada acrescida de c ao seu final)
      é maior que 1 e entrada+c não está no dicionário então
9:       grave no arquivo de saída o código associado à string armazenada em entrada
10:      insira entrada+c no dicionário, associando-a ao código novo_codigo
11:      novo_codigo ← novo_codigo + 1
12:      entrada ← c
13:   senão
14:      entrada ← entrada + c
15:   fim se
16:   leia um novo caractere do arquivo de entrada e armazene-o na variável c
17: fim enquanto
18: grave no arquivo de saída o código associado à string armazenada em entrada
19: feche os arquivos de entrada e saída
```

É importante ressaltar que o dicionário deve apenas conter entradas (*strings*) de comprimento maior ou igual a 2. Essa restrição aparece na linha 8 do algoritmo. Uma entrada de comprimento 1 é apenas um caractere e seu código já está definido na tabela ASCII.

2.1 Exemplo

Considere que temos como entrada para o algoritmo de compactação um arquivo de texto contendo apenas uma linha, com a seguinte sequência de caracteres

`as_asas_das_casas`

que corresponde a seguinte sequência de códigos ASCII

97 115 95 97 115 97 115 95 100 97 115 95 99 97 115 97 115

A tabela a seguir mostra cada um dos passos envolvidos na compactação desse arquivo de acordo com o Algoritmo 1.

	`c` = caractere lido do arquivo de entrada	`entrada`	`entrada+c`	inclui `entrada+c` no dicionário?	código gravado no arquivo de saída	novo valor de `entrada`
1	97 (a)		a	não		a
2	115 (s)	a	as	sim, código = 128	97 (a)	s
3	95 (_)	s	s_	sim, código = 129	115 (s)	_
4	97 (a)	_	_a	sim, código = 130	95 (_)	a
5	115 (s)	a	as	não		as
6	97 (a)	as	asa	sim, código = 131	128 (as)	a
7	115 (s)	a	as	não		as
8	95 (_)	as	as_	sim, código = 132	128 (as)	_
9	100 (d)	_	_d	sim, código = 133	95 (_)	d
10	97 (a)	d	da	sim, código = 134	100 (d)	a
11	115 (s)	a	as	não		as
12	95 (_)	as	as_	não		as_
13	115 (c)	as_	as_c	sim, código = 135	132 (as_)	c
14	97 (a)	c	ca	sim, código = 136	99 (c)	a
15	115 (s)	a	as	não		as
16	97 (a)	as	asa	não		asa
17	115 (s)	asa	asas	sim, código = 137	131 (asa)	s
18	—	s	—	—	115 (s)	—

De acordo com a sexta coluna dessa tabela, a sequência de códigos gravada no arquivo de saída e que representa de forma compactada o conteúdo do arquivo de texto de entrada é a seguinte:

97 115 95 128 128 95 100 132 99 131 115

É possível observar que na sequência de códigos do arquivo original tínhamos 17 elementos, enquanto na sequência "compactada" temos apenas 11. Entretanto, para determinar a taxa de compactação proporcionada pelo algoritmo, precisamos levar em conta a quantidade de bits necessária para representar cada um dos códigos antes e depois da compactação. Antes da compactação, os códigos que tínhamos eram somente os existentes na tabela ASCII, ou seja, valores entre -128 e 127, que ocupam 8 bits (1 byte) cada. Após a compactação, o maior código é o de número 132 – que precisa no mínimo de 9 bits para ser representado. Logo, no arquivo de entrada temos um consumo de $17 \times 8 = 136$ bits, enquanto que após a compactação, esse consumo pode ser de $11 \times 9 = 99$ bits.

3 Descompactação

O Algoritmo 2 descreve de forma estruturada os passos envolvidos na descompactação de um arquivo que foi compactado por meio do método de compactação LZW descrito na Seção 2. O algoritmo recebe como entrada um arquivo binário contendo uma sequência de códigos numéricos e gera como saída um arquivo de texto contendo os dados correspondentes aos códigos numéricos de forma descompactada.

Algoritmo 2 Descompactação

```
1: crie um dicionário vazio
2: abra o arquivo binário de entrada para leitura
3: abra o arquivo de texto de saída para escrita
4: novo_codigo ← 128 {primeiro código maior que os da tabela ASCII}
5: leia um código do arquivo de entrada e armazene-o na variável cod
6: entrada ← string associada ao código armazenado em cod
7: grave entrada no arquivo de saída
8: leia um novo código do arquivo de entrada e armazene-o na variável cod
9: enquanto não é o fim do arquivo de entrada faça
10:    entrada_anterior ← entrada
11:    entrada ← string associada ao código armazenado em cod
12:    grave entrada no arquivo de saída
13:    c ← primeiro caractere da string entrada
14:    insira entrada_anterior+c no dicionário, associando-a ao código novo_codigo
15:    novo_codigo ← novo_codigo + 1
16:    leia um novo código do arquivo de entrada e armazene-o na variável cod
17: fim enquanto
18: feche os arquivos de entrada e saída
```

Aqui é importante ressaltar que, nas linhas 6 e 11 do algoritmo, na obtenção da *string* associada a um dado código, é preciso levar em conta que esse código pode ser um código pertencente a uma entrada da tabela ASCII (nesse caso, o código estaria entre -128 e 127), ou um código referente a uma entrada do dicionário mantido pelo algoritmo (nesse caso, o código teria um valor maior ou igual a 128).

3.1 Exemplo

Considere que temos como entrada para o algoritmo de descompactação o arquivo binário gerado como saída do exemplo de compactação mostrado na Seção 2.1. A tabela a seguir mostra cada um dos passos envolvidos nessa descompactação de acordo com o Algoritmo 2.

	`cod` = código lido do arq. de entrada	`entrada_anterior`	`entrada`	`entrada_anterior` + 1º carac. de `entrada`	código inserido no dicionário	*string* gravada no arq. de saída
1	97		a			a
2	115	a	s	as	128	s
3	95	s	_	s_	129	_
4	128	_	as	_a	130	as
5	128	as	as	asa	131	as
6	95	as	_	as_	132	_
7	100	_	d	_d	133	d
8	132	d	as_	da	134	as_
9	99	as_	c	as_c	135	c
10	131	c	asa	ca	136	asa
11	115	asa	s	asas	137	s

4 O exercício programa

Você deverá escrever um programa em C que implemente a compactação/descompactação de arquivos de texto segundo o método LZW (descrito nos algoritmos das seções 2 e 3).

Podemos considerar que o dicionário usado nos algoritmos de compactação e descompactação é uma matriz de caracteres. Cada linha dessa matriz é um vetor de caracteres (= *string*) representando uma entrada armazenada no dicionário; o número de colunas da matriz indica o tamanho da maior entrada que pode ser armazenada no dicionário. Você deve considerar que o número máximo de entradas no seu dicionário é 30.000 e o tamanho máximo de uma entrada é 20.

Os algoritmos das seções 2 e 3 mencionam o termo "arquivo binário". No EP3, você aprendeu a ler arquivos de texto. Em um arquivo de texto, os dados são armazenados como sequências de caracteres divididas em linhas. Quando armazenamos um número em um arquivo de texto, cada um dos seus dígitos ocupa 1 byte (ou seja, o tamanho de uma variável do tipo *char*). Por exemplo, imagine que temos uma variável i do tipo *int* (que geralmente ocupa 4 bytes na memória do computador) e que o valor de i é 125896; gravar i em um arquivo de texto ocupa 6 bytes no arquivo. Já um arquivo binário pode conter qualquer tipo de dado; assim, se gravássemos i em um arquivo binário, gastaríamos os mesmos 4 bytes usados para manter a variável na memória do computador. Entretanto, não podemos abrir um arquivo binário em qualquer programa editor de textos (como o *Bloco de Notas* do Windows ou o *gedit* do Linux) e ver seu conteúdo diretamente, como fazemos com os arquivos .txt . Geralmente, um arquivo binário só faz sentido para o programa que o gerou, pois é preciso saber que tipos de dados foram gravados para que seja possível fazer a leitura correta dos dados.

Portanto, se quisermos ver os benefícios da compactação dos dados, não podemos usar um arquivo de texto para armazenar a sequência de códigos geradas pelo algoritmo de compactação. É por essa razão que você deverá usar um arquivo binário para armazenar os dados compactados. A Seção 5.2 explica como fazer as operações de abertura, leitura e escrita em arquivos binários.

Para garantir que os códigos associados às entradas do dicionário ocupem poucos bytes quando gravados no arquivo compactado, você deve sempre usar variáveis do tipo *short* para armazenar esses códigos em seu programa. Em EPs anteriores, vimos que o *long* é um tipo de dados para armazenar números inteiros com capacidade maior que o tipo *int*. Contrariamente, o *short* é um tipo de dados para armazenar números inteiros mas que possui capacidade menor que o *int*; um short geralmente ocupa 2 bytes na memória e é capaz de armazenar números no intervalo de -32.768 a 32.767 .

4.1 Interface com o usuário

O seu programa deve apresentar ao usuário um "menu" com as duas operações que podem ser feitas pelo programa, que são a compactação e a descompactação de arquivos de texto. Portanto, a aparência do menu mostrado ao usuário deve ser algo como:

```
Opcao 1 - Compactacao de um arquivo de texto
Opcao 2 - Descompactacao de um arquivo de texto
Opcao 3 - Sair do programa
Digite o numero da opcao desejada:
-
```

Após uma operação de compactação ou descompactação de arquivos bem sucedida, o seu programa deve sempre mostrar o menu ao usuário novamente. Em outras palavras: a execução do programa só deve terminar quando o usuário escolher a opção de sair do programa.

4.2 Operação de compactação: entrada e saídas

A entrada da operação de compactação é um arquivo de texto. Você deve pedir ao usuário o nome completo desse arquivo (lembrando que o nome completo de um arquivo inclui a sua extensão). O arquivo de

entrada deve estar na mesma pasta (= diretório) onde está localizado o executável do programa.

A operação de compactação deve gerar como saída dois arquivos:

- um arquivo binário contendo a sequência de códigos que representam os dados de entrada compactados (gerados por meio do Algoritmo 1);
- um arquivo de texto, que servirá apenas para verificar se o funcionamento do seu programa está correto. Esse arquivo deve conter as entradas criadas no dicionário na compactação e o seus respectivos códigos. Ele deve mostrar também os códigos gravados no arquivo binário e as suas respectivas *strings*. Esse arquivo é uma versão simplificada da tabela mostrada na Seção 2.1. O conteúdo do arquivo de texto gerado a partir do exemplo da Seção 2.1 é mostrado a seguir.

```
Arquivo Compactado              || Dicionario
Codigo | Saida                  || Codigo | Entrada
    97 | a                     ||    128 | as
   115 | s                     ||    129 | s_
    95 | _                     ||    130 | _a
   128 | as                    ||    131 | asa
   128 | as                    ||    132 | as_
    95 | _                     ||    133 | _d
   100 | d                     ||    134 | da
   132 | as_                   ||    135 | as_c
    99 | c                     ||    136 | ca
   131 | asa                   ||    137 | asas
   115 | s                     ||        |
```

Você deve pedir ao usuário um nome SEM a extensão para os arquivos de saída. Você deve usar como nome para a criação do arquivo binário o nome fornecido pelo usuário seguido pela extensão ".lzw". O nome do arquivo de texto gerado com os dados do dicionário deve ser o nome fornecido pelo usuário seguido pela extensão ".dic". Por exemplo, se o usuário fornecer "saida" como nome para os arquivos de saída, o seu programa deve criar dois arquivos nomeados de "saida.lzw" e "saida.dic". Os arquivos devem ser criados na mesma pasta onde está localizado o executável do programa.

Além do arquivo de texto de entrada do exemplo da Seção 2.1 (arquivo `ex1_asas.txt`), estão disponíveis na página da disciplina no sistema Paca (`http://paca.ime.usp.br/course/view.php?id=631`) mais dois arquivos que servem como exemplos de textos de entrada (`ex2_cartomante.txt` e `ex3_grimms.txt`). Use-os como entrada para o executável desse EP (que também está disponível no Paca) e olhe os arquivos gerados como saída. A taxa de compactação (= tamanho do arquivo compactado / tamanho do arquivo original) obtida pela aplicação do Algoritmo 1 sobre esses arquivos de entrada de exemplo pode ser vista na tabela a seguir. Pela tabela, podemos notar que a compressão no caso do arquivo `ex1_asas.txt` não foi vantajosa, uma vez que o arquivo compactado é maior que o original. Já no caso do exemplo 3, o compactação reduziu o arquivo quase à metade do seu tamanho original.

Arquivo de entrada	Tamanho do original	Tamanho do Compactado	Taxa de Compactação
`ex1_asas.txt`	17 bytes	22 bytes	129,4%
`ex2_cartomante.txt`	18,4kB	12,5kB	68%
`ex3_grimms.txt`	47,4kB	25,3kB	53,4%

4.3 Operação de descompactação: entrada e saídas

A entrada da operação de descompactação é um arquivo binário (gerado previamente por uma operação de compactação do seu programa). Você deve pedir ao usuário o nome completo desse arquivo (lembrando

que o nome completo de um arquivo inclui a sua extensão – que no caso deve ser ".lzw"). O arquivo de entrada deve estar na mesma pasta onde está localizado o executável do programa.

A operação de descompactação deve gerar como saída dois arquivos:

- um arquivo de texto contendo os dados descompactados (gerados por meio do Algoritmo 2). Se sua implementação da compactação e descompactação estiverem corretas, o conteúdo desse arquivo tem que ser exatamente igual ao conteúdo do arquivo usado como entrada para a operação de compactação;
- um arquivo de texto, que servirá apenas para verificar se o funcionamento do seu programa está correto. Esse arquivo deve conter as entradas criadas no dicionário na descompactação e o seus respectivos códigos. Ele deve mostrar também os códigos gravados no arquivo descompactado e as suas respectivas *strings*. Esse arquivo é uma versão simplificada da tabela mostrada na Seção 3.1. O conteúdo do arquivo de texto gerado a partir do exemplo da Seção 3.1 é mostrado a seguir.

```
Arquivo Descompactado             || Dicionario
Codigo | Saida                    || Codigo | Entrada
    97 | a                      ||        |
   115 | s                      ||    128 | as
    95 | _                      ||    129 | s_
   128 | as                     ||    130 | _a
   128 | as                     ||    131 | asa
    95 | _                      ||    132 | as_
   100 | d                      ||    133 | _d
   132 | as_                    ||    134 | da
    99 | c                      ||    135 | as_c
   131 | asa                    ||    136 | ca
   115 | s                      ||    137 | asas
```

Você deve pedir ao usuário um nome SEM a extensão para os arquivos de saída. Você deve usar como nome para a criação do arquivo de texto descompactado o nome fornecido pelo usuário seguido pela extensão ".txt". O nome do arquivo de texto gerado com os dados do dicionário deve ser o nome fornecido pelo usuário seguido pela extensão ".dic". Por exemplo, se o usuário fornecer "saida_desc" como nome para os arquivos de saída, o seu programa deve criar dois arquivos nomeados de "saida_desc.lzw" e "saida_desc.dic".

É importante observar que os algoritmos descritos nas seções 2 e 3 não incluem como saída a geração de um arquivo de texto com os dados do dicionário; eles somente geram um arquivo contendo os dados compactados ou descompactados. Você deve adaptar os algoritmos da forma que for necessário para incluir a geração desse segundo arquivo.

5 Manipulação de arquivos

5.1 Arquivos de texto

No EP3, você aprendeu a abrir um arquivo de texto para leitura usando o comando *fopen* e a ler dados do arquivo usando o comando *fscanf*. Neste EP, você precisará também criar um arquivo de texto para escrita e gravar dados nesse arquivo. Para isso, você usará as funções *fopen* e *fprintf*. O programa a seguir mostra um exemplo de gravação e leitura de um arquivo de texto. Esse exemplo contém todos os comandos de manipulação de arquivos de texto dos quais você precisará para fazer o seu EP.

```
#include <stdio.h>
#define TAM 10
```

```
/* Esse programa grava dados em um arquivo de texto e depois le esse arquivo
   caracter por caracter, imprimindo o seu conteudo na tela. */
int main()
{
    FILE *arq_entrada, *arq_saida;
    char nome_arquivo[20];

    short numeros[TAM] = {12, 23, 5, 9, 2, 1, 35, 8, 17, 26};
    int i;
    char c;

    printf("Digite o nome do arquivo: ");
    scanf("%s", nome_arquivo);

    /* Abre o arquivo de texto para escrita
       ("w" de write,) */
    arq_saida = fopen(nome_arquivo,"w");

    if (arq_saida == NULL)   /* Nao foi possivel criar/abrir o arquivo */
    {
        printf("ERRO: nao foi possivel criar o arquivo de saida \n");
        return -1;           /* Indica que houve erro */
    }

    /* Grava no arquivo uma string */
    fprintf(arq_saida, "O nome do arquivo criado e': %s \n", nome_arquivo);

    /* Grava no arquivo a sequencia de numeros inteiros
       que esta armazenada no vetor numeros */
    for (i = 0; i < TAM; i++)
        /* Grava no arquivo o numero armazenado em numeros[i] */
        fprintf(arq_saida, "%d ", numeros[i]);

    /*  Fecha o arquivo de saida */
    fclose(arq_saida);

    /* Abre para leitura o arquivo de texto criado acima
       ("r" de read)  */
    arq_entrada = fopen(nome_arquivo,"r");

    if (arq_entrada == NULL)   /* Nao foi possivel abrir o arquivo */
    {
        printf("ERRO: nao foi possivel abrir o arquivo de entrada \n");
        return -1;             /* Indica que houve erro */
    }

    /* Le um caracter do arquivo */
    fscanf(arq_entrada, "%c", &c);

    /* Enquanto o fim de arquivo (end of file) nao for encontrado,
       o laco imprime o ultimo caracter lido e le o proximo caracter.
       Antes de se chamar a funcao feof sobre um arquivo, e' sempre
       necessario tentar ler algo do arquivo, caso contrario
       o fim do arquivo pode nao ser detectado corretamente.
       Por isso, o comando fscanf apareceu antes desse while. */
    while (!feof(arq_entrada))
    {
        printf("%c", c);

        fscanf(arq_entrada, "%c", &c);
    }

    /*  Fecha o arquivo de entrada */
    fclose(arq_entrada);

    return 0;
}
```

Quando o comando `arq_entrada = fopen(nome_arquivo,"w")` é executado, um arquivo de texto vazio com nome `nome_arquivo` é criado para receber operações de gravação de dados. Se já existir um arquivo

com o nome indicado na mesma pasta do programa, então o conteúdo desse arquivo será descartado e o arquivo será tratado como um arquivo novo.

5.2 Arquivos binários

Neste EP, além de manipular arquivos de texto, você precisará também manipular arquivos binários. Assim como feito com os arquivos de texto, a abertura/criação de arquivos binários é feita por meio da função *fopen*, mas com parâmetros diferentes. As operações de leitura e escrita no arquivo são feitas por meio dos comandos *fread* e *fwrite*, respectivamente. O programa a seguir mostra um exemplo de gravação e leitura de um arquivo binário. Esse exemplo contém todos os comandos de manipulação de arquivos binários dos quais você precisará para fazer o seu EP.

```
#include <stdio.h>
#define TAM 10

/* Esse programa grava uma sequencia de numeros em um arquivo
   binario e depois le esse arquivo imprimindo na tela os numeros. */
int main()
{
    FILE *arq_entrada, *arq_saida;
    char nome_arquivo[20];

    short numeros[TAM] = {12, 23, 5, 9, 2, 1, 35, 8, 17, 26};
    short num_arq;
    int i;

    printf("Digite o nome do arquivo: ");
    scanf("%s", nome_arquivo);

    /* Abre o arquivo binario para escrita
       ("w" de write, "b" de binary) */
    arq_saida = fopen(nome_arquivo,"wb");

    if (arq_saida == NULL)   /* Nao foi possivel criar/abrir o arquivo */
    {
        printf("ERRO: nao foi possivel criar o arquivo de saida \n");
        return -1;           /* Indica que houve erro */
    }

    for (i = 0; i < TAM; i++)
        /* Grava no arquivo o numero armazenado em numeros[i] */
        fwrite(&numeros[i], sizeof(short), 1, arq_saida);

    /*  Fecha o arquivo de saida */
    fclose(arq_saida);

    /* Abre para leitura o arquivo binario criado acima
       ("r" de read, "b" de binary)  */
    arq_entrada = fopen(nome_arquivo,"rb");

    if (arq_entrada == NULL)   /* Nao foi possivel abrir o arquivo */
    {
        printf("ERRO: nao foi possivel abrir o arquivo de entrada \n");
        return -1;             /* Indica que houve erro */
    }

    /* Le um numero do arquivo */
    fread(&num_arq, sizeof(short), 1, arq_entrada);

    /* Enquanto o fim de arquivo (end of file) nao foi encontrado,
       o laco imprime o numero lido e depois le um novo numero do
       arquivo.
       Antes de se chamar a funcao feof sobre um arquivo, e' sempre
       necessario tentar ler algo do arquivo, caso contrario
       o fim do arquivo pode nao ser detectado corretamente.
       Por isso, o comando fread apareceu antes desse while. */
    while (!feof(arq_entrada))
```

```
    {
        printf("%d ", num_arq);

        fread(&num_arq, sizeof(short), 1, arq_entrada);
    }

    /*  Fecha o arquivo de entrada */
    fclose(arq_entrada);

    return 0;
}
```

Quando o comando `arq_entrada = fopen(nome_arquivo,"wb")` é executado, um arquivo binário vazio com nome `nome_arquivo` é criado para receber operações de gravação de dados. Se já existir um arquivo com o nome indicado na mesma pasta do programa, então o conteúdo desse arquivo será descartado e o arquivo será tratado como um arquivo novo.

Como pode ser visto no exemplo, a função de gravação *fwrite* recebe 4 parâmetros de entrada:

1. um ponteiro para a dado que será gravado (= o endereço da variável que armazena o dado);
2. o tamanho em bytes de um elemento a ser gravado. Geralmente, o valor para esse parâmetro é obtido por meio da função *sizeof* (que recebe como entrada um tipo de dados e devolve o seu tamanho em bytes);
3. o número de elementos que serão gravados (cada um deles deve possuir como tamanho o número indicado no parâmetro anterior);
4. o arquivo onde os dados serão gravados.

Portanto, o comando `fwrite(&numeros[i], sizeof(short), 1, arq_saida)` que aparece no exemplo grava em `arq_saida` somente 1 elemento – o valor de `numeros[i]` – que ocupa `sizeof(short)` bytes.

Analogamente, a função de leitura *fread* também possui 4 parâmetros (sendo o primeiro deles de saída): (1) o endereço da variável que armazenará o dado lido; (2) o tamanho em bytes do elemento a ser lido; (3) o número de elementos que serão lidos; (4) o arquivo de onde os dados serão lidos. Portanto, o comando `fread(&num_arq, sizeof(short), 1, arq_entrada)` que aparece no exemplo lê do arquivo `arq_entrada` somente 1 elemento de `sizeof(short)` bytes e armazena-o na variável `num_arq`.

6 Observações Finais

- Neste EP, não foram definidos protótipos de funções a serem obrigatoriamente implementadas. Entretanto, um dos critérios de avaliação do EP será a organização do código. Portanto, sempre que for conveniente, divida o seu código em funções. Deixar todo o código na função *main* compromete a legibilidade do seu programa e (o mais grave!) deixa-o mais susceptível a erros.
- O uso de funções da biblioteca `string.h` está liberado. Aliás, o uso é aconselhado!
- O seu programa não precisa funcionar para arquivos de entrada grandes (com mais de 50kB). Para arquivos grandes, seria preciso um dicionário com mais de 30.000 entradas e, consequentemente, códigos que passariam dos 2 bytes (que é o tamanho de uma variável do tipo *short*).
- O algoritmo de descompactação descrito na Seção 3 não funciona quando o arquivo de entrada contém uma sequência de 3 caracteres ou *strings* repetidos (como "rrr" ou "aiaiai"). Portanto, não use em seus testes sequências de caracteres com essas características.