Capítulo 12

ALOCAÇÃO DINÂMICA DE MEMÓRIA

12.12 Exemplos de Programação

12.12.1 Lendo Linhas (Praticamente) Ilimitadas

Problema: (a) Escreva uma função que lê linhas de tamanhos arbitrários num stream de texto (inclusive **stdin**) e converte-as num string que não contenha o caractere de quebra de linha (`'\n'`). (b) Escreva um programa que leia e apresente cada linha de um arquivo de texto e apresente-a na tela. Esse programa deve ainda ler um string introduzido via teclado e apresentá-lo na tela.

Solução do item (a):

```
/****
 *
 * LeLinhaIlimitada(): Lê uma linha de tamanho arbitrário num
 *                     stream de texto e armazena os caracteres
 *                     lidos num array alocado dinamicamente
 *
 * Parâmetros:
 *      tam (saída) - se não for NULL, apontará para uma variável
 *                    que armazenará o tamanho do string
 *                    constituído pelos caracteres da linha
 *      stream (entrada) - stream de texto no qual será feita a
 *                         leitura
 *
 * Retorno: Endereço do array contendo a linha lida. NULL, se
 *          ocorrer erro ou o final do arquivo for atingido
 *          antes da leitura de qualquer caractere
 *
 * Observações:
 *       1. O stream deve estar associado a um arquivo de texto
 *          aberto em modo de texto que permita leitura
 *       2. O caractere '\n' não é incluído no string resultante
 *          da leitura
 *       3. O primeiro parâmetro pode ser NULL. Nesse caso, o
 *          tamanho do string não será armazenado
 *
 ****/
char *LeLinhaIlimitada(int *tam, FILE *stream)
{
   char *ar = NULL,   /* Ponteiro para um array alocado */
                      /* dinamicamente que conterá os   */
                      /* caracteres lidos               */
        *p; /* Usado em chamada de realloc() */
   int   tamanho = 0, /* Tamanho do array alocado */
```

```
      c,  /* Armazenará cada caractere lido */
      i; /* Índice do próximo caractere */
         /* a ser inserido no array     */

   /* Lê caracteres a partir da posição corrente do      */
   /* indicador de posição do arquivo e armazena-os num  */
   /* array. A leitura encerra quando '\n' é encontrado, */
   /* o final do arquivo é atingido ou ocorre erro.      */
for (i = 0; ; ++i) {
      /* Lê o próximo caractere no arquivo */
   c = fgetc(stream);

      /* Se ocorreu erro de leitura, libera o  */
      /* bloco eventualmente alocado e retorna */
   if (ferror(stream)) {
      free(ar); /* Libera o bloco apontado por 'ar' */
      return NULL; /* Ocorreu erro de leitura */
   }

      /* Verifica se array está completo. O maior valor que i*/
      /* poderia assumir deveria ser tamanho - 1. Mas, como  */
      /* ao final, o caractere '\0' deverá ser inserido,     */
      /* limita-se o valor de i a tamanho - 2.               */
   if (i > tamanho - 2) { /* Limite atingido */
         /* Tenta redimensionar o array */
      p = realloc(ar, tamanho + TAMANHO_BLOCO);

         /* Se o redimensionamento não foi     */
         /* possível, libera o bloco e retorna */
      if (!p) {
         free(ar); /* Libera o bloco apontado por 'ar' */
         return NULL; /* Ocorreu erro de alocação */
      }

         /* Redimensionamento foi OK. Então, faz-se */
         /* 'ar' apontar para o novo bloco.         */
      ar = p;

         /* O array aumentou de tamanho */
      tamanho = tamanho + TAMANHO_BLOCO;
   }

      /* Se final do arquivo for atingido ou o caractere */
      /* '\n' for lido,  encerra-se a leitura            */
   if (feof(stream) || c == '\n') {
      break; /* Leitura encerrada */
   }

   ar[i] = c; /* Acrescenta o último caractere lido ao array */
}

   /* Se nenhum caractere foi lido, libera */
   /* o espaço alocado e retorna NULL      */
if (feof(stream) && !i) {
   free(ar); /* Libera o bloco apontado por 'ar' */
   return NULL; /* Nenhum caractere foi armazenado no array */
}

   /* Insere o caractere terminal de string. Neste    */
   /* instante, deve haver espaço para ele porque o   */
   /* array foi sempre redimensionado deixando um     */
   /* espaço a mais para o onipresente caractere '\0' */
ar[i] = '\0';
```

```
      /* Atualiza o valor apontando pelo parâmetro */
      /* 'tam', se ele não for NULL                */
   if (tam) {
         /* i é o índice do caractere terminal do */
         /* string e corresponde ao seu tamanho   */
      *tam = i;
   }

            /*                                     */
            /* >>> NB: O tamanho do string não <<< */
            /* >>>     inclui o caractere '\0' <<< */
            /*                                     */

      /* Tenta ajustar o tamanho do array para não */
      /* haver desperdício de memória. Como i é o  */
      /* tamanho do string, o tamanho do array que */
      /* o contém deve ser i + 1.                  */
   p = realloc(ar, i + 1);

      /*                                                */
      /* Se a realocação foi bem sucedida, retorna-se p. */
      /* Caso contrário, 'ar' ainda aponta para um bloco */
      /* válido. Talvez, haja desperdício de memória,    */
      /* mas, aqui, é melhor retornar 'ar' do que NULL.  */
      /*                                                */
   return p ? p : ar;
}
```

Análise: Antes de tentar entender o funcionamento da função `LeLinhaIlimitada()`, que é relativamente complexo, é essencial que você assimile bem o que essa função exatamente faz. Com esse propósito, observe na **Tabela 1** a comparação entre essa função e a função **fgets()**, discutida na **Seção 10.9.6**.

fgets()	`LeLinhaIlimitada()`
Lê caracteres num stream de texto a partir do local corrente do indicador de posição de arquivo, até encontrar `'\n'`, o final do arquivo ou ocorrer erro	Idem
Armazena os caracteres lidos num array e acrescenta o caractere `'\0'` ao final dos caracteres lidos	Idem
Quando encontra um caractere `'\n'`, ele é armazenado no array	Não armazena caractere `'\n'`
Retorna **NULL** quando nenhum caractere é lido	Retorna **NULL** quando ocorre erro de leitura ou de alocação dinâmica, mesmo que algum caractere tenha sido lido
O array no qual os caracteres são armazenados é recebido como	O array no qual os caracteres são armazenados é alocado dinamicamente

parâmetro	
O número de caracteres lidos é limitado por um parâmetro que indica o tamanho do array	O número de caracteres lidos é limitado pelo espaço disponível no heap, o que, em condições normais, significa que não há limite para o número de caracteres lidos
Não informa o tamanho do string resultante de uma leitura	O tamanho do string resultante de uma leitura é armazenado numa variável por meio do primeiro parâmetro da função, se esse parâmetro não for **NULL**

Tabela 1: Comparação entre fgets() e LeLinhaIlimitada()

É importante salientar que, como `LeLinhaIlimitada()` aloca espaço dinamicamente, cada chamada dessa função deve ser emparelhada com uma chamada de **free()** para liberar o espaço alocado para a linha lida quando esta deixa de ser necessária.

Agora que você já conhece bem o que a função `LeLinhaIlimitada()` faz, prepare-se, pois essa função possui muitos detalhes importantes que serão explorados a seguir.

- É importante chamar atenção para os importantes papéis desempenhados pelas variáveis locais `ar`, `i` e `tamanho`:
    - A variável `ar`, que é iniciada com **NULL**, *quase sempre* aponta para o array alocado dinamicamente que armazena os caracteres que irão compor o string. Existe um único instante em que essa variável pode não apontar para esse array, que é logo após uma tentativa de seu redimensionamento.
    - Em qualquer instante, o valor da variável `i` indica o índice do próximo caractere a ser inserido no array. Essa variável é iniciada com zero no laço **for** da função em discussão.
    - A variável `tamanho` sempre armazena o tamanho (i.e., número de bytes) do referido array e é iniciada com zero.

  As demais variáveis locais, `c` e `p`, têm papéis secundários, que serão facilmente entendidos mais adiante.
- A leitura da linha é efetuada no laço **for** da função supracitada. Esse laço, deve-se frisar, não tem condição natural de parada, pois seu encerramento acontecerá no corpo do mesmo laço. Mais precisamente, o laço **for** termina quando ocorre erro de leitura, tentativa de leitura além do final do arquivo ou quando o caractere `'\n'` é encontrado.

- A primeira instrução no corpo do aludido laço **for** lê um caractere no arquivo por meio de **fgetc()**:

```
c = fgetc(stream);
```

- A próxima instrução do corpo do mesmo laço testa se ocorreu erro de leitura, e, se esse for o caso, o array é liberado, por meio de uma chamada de **free()**. Então, a função em discussão retorna **NULL**, indicando que a leitura foi mal sucedida.

```
if (ferror(stream)) {
   free(ar);
   return NULL;
}
```

  Note que, se não houve espaço alocado (i.e., se ocorreu erro na primeira tentativa de leitura), não haverá problema com a referida chamada de **free()**, porque o ponteiro usado como parâmetro nessa chamada foi iniciado com **NULL**.

- A próxima instrução no corpo do laço **for** é uma instrução **if** que tenta redimensionar o array que armazenará a linha lida quando a seguinte condição é satisfeita:

```
i > tamanho - 2
```

  Nessa expressão, `i` é o índice do próximo caractere a ser inserido no array e `tamanho` é o número de elementos desse array. O que justifica essa expressão é o fato de, antes de inserir o último caractere lido no array, ser necessário haver espaço livre no array para, pelo menos, mais dois caracteres: o último caractere lido e o caractere terminal de string (`'\0'`). Ou, dito de outro modo, se o número de caracteres armazenados no array for maior do que o tamanho corrente do array menos dois, será necessário redimensionar o array. Ora, mas como `i` indica o índice do próximo caractere a ser armazenado no array, o valor dessa variável também corresponde ao número de caracteres correntemente armazenados no array Logo, como o tamanho corrente do array é representado pela variável `tamanho`, pode-se escrever em C a condição para que o redimensionamento do array seja necessário como: `i > tamanho - 2`, que é a expressão da instrução **if** em questão.

- No corpo da última instrução **if**, a primeira instrução é responsável pelo redimensionamento mencionado no parágrafo anterior:

```
p = realloc(ar, tamanho + TAMANHO_BLOCO);
```

  Nessa instrução, faz-se uma tentativa de redimensionamento do array por meio de uma chamada de **realloc()**. Nessa chamada, tenta-se acrescer um

número de bytes igual à constante simbólica `TAMANHO_BLOCO` ao tamanho do array[1]. Deve-se notar que, conforme foi recomendado na **Seção 12.2.4**, o valor retornado por **realloc()** foi atribuído ao ponteiro local `p` (e não ao ponteiro `ar`) para evitar que o bloco apontado por `ar` seja definitivamente perdido (v. adiante).

A próxima instrução no corpo do laço testa se alocação foi mal sucedida e, se esse for o caso, libera-se o espaço alocado anteriormente para o array e retorna-se **NULL**, indicando que a função não foi bem sucedida.

```
if (!p) {
   free(ar);
   return NULL;
}
```

Ainda nesse caso, observe que, se o resultado retornado por **realloc()** tivesse sido atribuído a `ar`, a liberação do array não seria possível.

As duas últimas instruções do corpo da instrução **if** que realiza o redimensionamento são executada apenas quando a realocação do array é bem sucedida:

```
ar = p;
```

```
tamanho = tamanho + TAMANHO_BLOCO;
```

A primeira dessas instruções faz `ar` apontar novamente para o array que conterá o resultado da leitura, enquanto que a segunda instrução atualiza o valor da variável `tamanho` para que ela reflita o novo tamanho do array.

- Após o eventual redimensionamento do array, verifica-se se a leitura deve ser encerrada por meio da instrução **if**:

```
if (feof(stream) || c == '\n') {
   break;
}
```

Nessa instrução **if**, duas condições encerram a execução do laço **for**: tentativa de leitura além do final do arquivo ou leitura de uma quebra de linha (`'\n'`).

- A última instrução no corpo do laço **for** acrescenta o último caractere lido ao array:

```
ar[i] = c;
```

[1] Aparentemente, uma boa idéia seria aumentar o tamanho do array a cada caractere lido, pois, assim, não haver nenhum desperdício de memória. Mas, de fato, essa não é uma boa alternativa devido ao ônus associado a cada chamada de **realloc()** (v. **Seção 12.2.4** e início da **Seção 12.10**).

Evidentemente, se o último caractere lido foi `'\n'`, ele não será inserido no array, porque, nesse caso, o laço **for** já terá sido encerrado pela instrução **if** anterior.

- A primeira instrução após o laço **for** verifica se nenhum caractere foi lido e, se esse for o caso, o array é liberado e a função retorna **NULL**.

```
if (feof(stream) && !i) {
   free(ar);
   return NULL;
}
```

Essa instrução **if** contém uma sutileza que talvez passe desapercebida. Quer dizer, aparentemente, não seria necessário testar se o final do arquivo foi atingido, pois seria suficiente checar se algum caractere foi armazenado no array (i.e., verificar se o valor de `i` é igual a zero). Mas, lembre-se que, quando o caractere `'\n'` é lido, ele não é armazenado no array. Concluindo, se a chamada de **feof()** fosse removida da expressão condicional da referida instrução **if**, a função não seria capaz de ler linhas vazias (i.e., linhas contendo apenas `'\n'`).

- Se ainda não houve retorno da função, pelo menos um caractere foi lido, mesmo que ele não tenha sido armazenado no array. Então, acrescenta-se o caractere terminal ao array na posição indicada por `i` para que o array contenha um string:

```
ar[i] = '\0';
```

- Como, nesse instante, `i` é o índice do caractere terminal do string armazenado no array e a indexação de arrays começa com zero, o valor dessa variável corresponde exatamente ao tamanho do string (sem incluir o caractere `'\0'`, como usual). Logo, se o primeiro parâmetro não for **NULL**, o valor de `i` é atribuído ao conteúdo apontado por ele, como faz a instrução **if** a seguir:

```
if (tam) {
   *tam = i;
}
```

- Para evitar desperdício de memória, tenta-se ajustar o tamanho do array para que esse tamanho seja exatamente igual ao número de caracteres armazenados no array (incluindo `'\0'`), que é obtido avaliando-se a expressão `i + 1` na chamada de **realloc()**:

```
p = realloc(ar, i + 1);
```

- Se a realocação foi bem sucedida, a função retorna o valor retornado por **realloc()** e armazenado em `p`. Caso contrário, é retornado o valor de `ar`, que ainda aponta para um bloco válido.

```
                                    return p ? p : ar;
```

Quando a última chamada de **realloc()** não é bem sucedida, é possível que haja desperdício de memória, mas, aqui, é bem mais sensato retornar `ar` do que **NULL**.

Solução do item (b):

```
/****
 *
 * main(): Lê linhas de tamanho arbitrário num arquivo de texto e
 *         via teclado, e apresenta-as na tela
 *
 * Parâmetros: Nenhum
 *
 * Retorno: 0, se não ocorrer nenhum erro; 1, em caso contrário.
 *
 ****/
int main(void)
{
   FILE *stream;
   char *linha; /* Apontará para cada linha lida */
   int   tamanho,  /* Tamanho de cada linha lida */
         nLinhas = 0; /* Número de linhas do arquivo */

      /* Tenta abrir para leitura em modo texto o arquivo */
      /* cujo nome é dado pela constante NOME_ARQ         */
   stream = fopen(NOME_ARQ, "r");

      /* Se o arquivo não foi aberto, encerra o programa */
   if (!stream) {
      printf("\n\t>>> Arquivo nao pode ser aberto\n");
      return 1; /* Arquivo não foi aberto */
   }

            /*                                        */
            /* Lê o conteúdo do arquivo linha a linha */
            /* informando o tamanho de cada linha     */
            /*                                        */

   printf("\n\t\t*** Conteudo do Arquivo ***\n");

      /* O laço encerra quando 'linha' assumir  */
      /* NULL, o que acontece quando todo o     */
      /* arquivo for lido ou ocorrer algum erro */
   while ( (linha = LeLinhaIlimitada(&tamanho, stream)) ) {
         /* Escreve o número da linha */
      printf("\n>>> Linha %d: ", nLinhas + 1);
         /* Apresenta a linha seguida por seu tamanho */
      printf("%s (%d caracteres)\n", linha, tamanho);

      free(linha); /* Libera o espaço ocupado pela linha */

      ++nLinhas; /* Mais uma linha foi lida */
   }

      /* Informa quantas linhas foram lidas no arquivo */
   printf("\n\t>>> O arquivo possui %d linhas\n", nLinhas);
```

```
      /* Fecha-se o arquivo, pois ele não é mais necessário */
   fclose(stream);

         /*                                              */
         /* Lê um string de tamanho ilimitado em stdin */
         /*                                              */

   printf("\n\t>>> Digite um texto de qualquer tamanho:\n\t> ");
   linha = LeLinhaIlimitada(&tamanho, stdin);

   printf("\n\t>>> Texto introduzido:\n\"%s\"\n", linha);
   printf( "\n\t>>> Tamanho do texto digitado: %d caracteres\n",
           tamanho );

   free(linha); /* Libera espaço ocupado pelo string lido */

   return 0;
}
```

Análise: Essa função **main()** é fácil de entender, mas o leitor deve atentar para o fato de cada chamada da função `LeLinhaIlimitada()` ser emparelhada com uma chamada de **free()** para evitar escoamento de memória (v. **Seção 12.4**).

Complemento do programa:

```
#include <stdio.h>   /* Entrada e saída   */
#include <stdlib.h>  /* Alocação dinâmica */

   /* Nome do arquivo usado nos testes do programa */
#define NOME_ARQ      "AnedotaBulgara.txt"

   /* Tamanho do acréscimo do bloco usado para conter */
   /* uma linha a cada chamada de realloc()           */
#define TAMANHO_BLOCO 256
```

Exemplo de execução do programa:

```
                *** Conteudo do Arquivo ***

>>> Linha 1: Anedota Bulgara (15 caracteres)

>>> Linha 2: Carlos Drummond de Andrade (26 caracteres)

>>> Linha 3:  (0 caracteres)

>>> Linha 4: Era uma vez um czar naturalista (31 caracteres)

>>> Linha 5: que cacava homens. (18 caracteres)

>>> Linha 6: Quando lhe disseram que tambem se (33 caracteres)

>>> Linha 7: cacam borboletas e andorinhas, (30 caracteres)

>>> Linha 8: ficou muito espantado (21 caracteres)

>>> Linha 9: e achou uma barbaridade (23 caracteres)

        >>> O arquivo possui 9 linhas
```

```
        >>> Digite um texto de qualquer tamanho:
        > Pedro de Alcantara Francisco Antonio Joao Carlos
Xavier de Paula Miguel Rafael Joaquim Jose Gonzaga Pascoal
Cipriano Serafim de Braganca e Bourbon

        >>> Texto introduzido:
"Pedro de Alcantara Francisco Antonio Joao Carlos Xavier de
Paula Miguel Rafael Joaquim Jose Gonzaga Pascoal Cipriano
Serafim de Braganca e Bourbon"

        >>> Tamanho do texto digitado: 146 caracteres
```