Capítulo 12

Alocação Dinâmica de Memória

Após estudar este capítulo, você deverá ser capaz de:

- Definir e usar os seguintes conceitos:
  - ❒ Variável estática
  - ❒ Variável dinâmica
  - ❒ Variável anônima
  - ❒ Partição heap
  - ❒ Ponteiro genérico
  - ❒ Zumbi de heap
  - ❒ Array estático
  - ❒ Array dinâmico
  - ❒ Alocação estática
  - ❒ Alocação dinâmica
  - ❒ Subdimensionamento de memória
  - ❒ Superdimensionamento de memória
- Descrever o funcionamento das seguintes funções da biblioteca padrão de C:
  - ❒ **malloc()**
  - ❒ **calloc()**
  - ❒ **realloc()**
  - ❒ **free()**
- Exibir situações em programação nas quais alocação dinâmica de memória se faz necessária
- Descrever o funcionamento das funções de alocação dinâmica de memória de C
- Explicar por que não se deve atribuir o valor retornado por **realloc()** ao mesmo ponteiro passado como primeiro parâmetro numa chamada dessa função
- Prevenir-se contra erros comuns de liberação de blocos alocados dinamicamente
- Testar o endereço retornado por uma função de alocação dinâmica de memória
- Mostrar o papel desempenhado por **realloc()** no processamento de arrays dinâmicos
- Implementar uma função capaz de ler linhas de qualquer tamanho num stream de texto
- Explicar como tipicamente é dividido o espaço reservado para a execução de um programa
- Identificar os sintomas aparentes de um programa com escoamento de memória

OBJETIVOS

12.1 Introdução

UMA VARIÁVEL DE DURAÇÃO FIXA tem memória reservada para si durante todo o tempo de execução do programa que a utiliza, enquanto uma variável de duração automática é alocada cada vez que o bloco que a contém é executado (v. **Seção 5.9**). Em ambas as formas de alocação de memória, assume-se que, durante a escrita de um programa, o programador sabe qual é a quantidade de memória necessária para sua execução. Entretanto, existem muitas situações nas quais a quantidade de memória necessária para armazenar os dados de um programa não pode ser determinada precisamente em tempo de programação. Por exemplo, suponha que um programa precisa ler num arquivo de texto uma linha de tamanho arbitrário e armazená-la num array para posterior processamento. Mas, se o tamanho da linha é desconhecido, como o programador deverá proceder para dimensionar o array que armazenará a linha?

Com o conhecimento adquirido até aqui, o melhor que o programador pode fazer é definir uma constante simbólica que representa uma estimativa de tamanho da linha e, então, definir o array que armazenará essa linha utilizando essa constante como mostra o fragmento de programa abaixo:

```
#define MAIOR_LINHA 200
...
char linha[MAIOR_LINHA];
```

Ocorre que, usando essa abordagem de solução, surgem dois novos problemas potenciais:

- **Subdimensionamento**. Nesse caso, o tamanho estimado para a linha é menor do que o tamanho real da linha e essa linha não poderá ser armazenada no array sem haver corrupção de memória.
- **Superdimensionamento**. Aqui, o tamanho estimado para a linha é maior do que o tamanho real da linha. Nesse caso, a linha poderá ser armazenada no array com folga, mas, dependendo de quanto é essa *folga*, poderá haver grande desperdício de memória.

Vários programas apresentados como exemplos em capítulos anteriores sofrem com essa falta de capacidade de alocar memória precisamente. Em todos eles ocorre superdimensionamento (se ocorresse subdimensionamento, como eles iriam funcionar?). Por exemplo, o programa apresentado na **Seção 11.15.9** usa duas constantes simbólicas: uma para estimar o tamanho máximo de uma linha do arquivo que ele lê e a outra para estimar o número máximo de caracteres num nome. Nesse exemplo, o programa ora pode desperdiçar memória, em virtude de superdimensionamento, ora pode deixar de funcionar adequadamente, por causa de subdimensionamento.

O que deve ter ficado claro nos exemplos mencionados é que os programas em questão precisam ser dotados de capacidade para alocar memória de acordo com a demanda apresentada durante suas execuções.

A melhor solução para problemas nos quais a memória necessária para execução de um programa não pode ser precisamente estimada é a alocação de memória decidida durante a execução do programa e de acordo com a demanda manifestada. Esse tipo de alocação é denominado **alocação dinâmica de memória** e contrasta com qualquer outro tipo de alocação de memória visto até aqui, denominado **alocação estática de memória**, cujo espaço a ser alocado é conhecido em tempo de programação. Variáveis alocadas estaticamente são denominadas **variáveis estáticas**, enquanto variáveis alocadas dinamicamente são denominadas **variáveis dinâmicas**. Nesse sentido, todas as variáveis vistas até aqui são estáticas. O objetivo central deste capítulo é mostrar como variáveis podem ser alocadas dinamicamente.

12.2 Funções de Alocação Dinâmica de Memória

Alocação dinâmica de memória em C é realizada por meio de ponteiros e quatro funções de biblioteca resumidas na **Tabela 12–1**. Para utilizar essas funções, inclua em seu programa o cabeçalho `<stdlib.h>`.

Função	Descrição Resumida
malloc()	Aloca um número especificado de bytes em memória e retorna o endereço inicial do bloco de memória alocado. O conteúdo do bloco alocado é indefinido.
calloc()	Essa função é similar a **malloc()**, mas adicionalmente, ela inicia todos os bytes alocados com zeros e também permite a alocação de um array de blocos.
realloc()	Altera o tamanho de um bloco previamente alocado dinamicamente.
free()	Libera o espaço ocupado por um bloco de memória previamente alocado com **malloc()**, **calloc()** ou **realloc()**.

Tabela 12–1: Funções de Alocação Dinâmica de Memória

Essas funções serão exploradas em profundidade adiante, mas, antes de prosseguir, é oportuno relembrar o conceito de bloco de memória[1], que é um conjunto de bytes contíguos em memória (v. Seção 11.11.3). O tipo **size_t** (v. Seção 8.5) é utilizado pelas funções de alocação de memória para especificar tamanhos de blocos de memória (v. adiante).

Blocos alocados dinamicamente são às vezes referidos como **variáveis anônimas**, pois eles têm conteúdo e endereço, como variáveis comuns, mas não têm nome e, por isso, seus conteúdos podem ser acessados apenas indiretamente por meio de ponteiros.

12.2.1 malloc()

A função **malloc()**, cujo protótipo é apresentado a seguir, recebe como parâmetro o tamanho, em bytes, do bloco a ser dinamicamente alocado e retorna o endereço inicial desse bloco, se ele for efetivamente alocado.

```
void *malloc(size_t tamanho)
```

Usualmente, o parâmetro real recebido por essa função envolve o uso do operador **sizeof**, que é recomendado, principalmente, por questões de praticidade e portabilidade. Por exemplo, supondo que `ptrAluno` seja um ponteiro para o tipo `tAluno`, definido na Seção 11.15.10, então, a seguinte chamada da função **malloc()**:

```
ptrAluno = malloc(sizeof(tAluno));
```

alocaria (se fosse possível) um bloco capaz de conter uma estrutura do tipo `tAluno` e retornaria o endereço inicial desse bloco.

De acordo com o protótipo acima, o tipo de retorno de **malloc()** é **void** *, que representa um tipo que será discutido na Seção 12.3. Esse tipo permite que o valor retornado por **malloc()** possa ser atribuído a qualquer ponteiro sem que seja necessário o uso de conversão explícita, conforme mostra o último exemplo.

Quando a função **malloc()** não consegue alocar espaço em memória para o bloco requerido, ela retorna **NULL** (v. Seção 12.4).

12.2.2 calloc()

A função **calloc()** recebe dois parâmetros: o primeiro é o número de blocos a serem alocados e o segundo é o tamanho de cada bloco. Seu protótipo é:

```
void *calloc(size_t nBlocos, size_t tamanho)
```

Quando possível, a função **calloc()** aloca o espaço necessário para conter os blocos requisitados e retorna o endereço inicial do primeiro bloco alocado. Todos os bytes do espaço alocado são iniciados com zeros.

[1] Doravante, neste capítulo, *bloco* será utilizado como sinônimo de *bloco de memória*.

Apesar de não ser necessário, é instrutivo examinar como a função **calloc()** poderia ser implementada utilizando **malloc()**. A função `MinhaCalloc()` apresentada a seguir é funcionalmente equivalente a **calloc()**.

```
void *MinhaCalloc(size_t nBlocos, size_t tamanho)
{
   size_t  i, nBytes;
   char   *ptr;

      /* Calcula o número de bytes que serão alocados */
   nBytes = nBlocos*tamanho;

   ptr = malloc(nBytes); /* Tenta alocar o bloco requisitado */

   if (!ptr) { /* Checa se houve alocação */
      return NULL; /* Não foi possível alocar o bloco */
   }

      /* Aqui, o espaço já foi alocado. Resta apenas zerar os bytes. */
   for (i = 0; i < nBytes; ++i) {
      ptr[i] = 0; /* Zera cada byte */
   }

   return ptr; /* Trabalho completo */
}
```

Para entender a implementação da função `MinhaCalloc()`, note que essa função deve alocar um número de blocos determinado pelo parâmetro `nBlocos` e que o tamanho de cada bloco é especificado pelo parâmetro `tamanho`. Ora, mas isso é equivalente a alocar um único bloco cujo tamanho é dado por:

```
nBlocos*tamanho
```

uma vez que não apenas os bytes de cada bloco são contíguos em memória como também todos os blocos devem ser contíguos. A variável local `nBytes` é utilizada para conter esse valor e, embora não seja estritamente necessária, ela é utilizada como um fator de otimização da função (v. adiante).

O ponteiro `ptr`, que representa o valor retornado por **malloc()** e que será posteriormente retornado pela função `MinhaCalloc()`, é definido com o tipo **char** *, porque ele também será utilizado com o objetivo de zerar cada byte do bloco alocado. O objetivo do laço **for** da função `MinhaCalloc()` é exatamente realizar a tarefa de zerar cada byte do bloco. Esse laço também justifica o uso da variável `nBytes`; i.e., se essa variável não fosse utilizada, o produto `nBlocos*tamanho` teria que ser calculado a cada avaliação da expressão condicional desse laço.

12.2.3 free()

A função **free()** recebe como único parâmetro um ponteiro que aponta para um bloco de memória alocado utilizando **malloc()**, **calloc()** ou **realloc()** e libera o espaço ocupado pelo bloco, de forma que ele se torna disponível para futuras alocações. Se o ponteiro passado para **free()** for nulo, a função retorna sem executar nada.

O protótipo da função **free()** é:

```
void free(void *ptr)
```

Após uma chamada da função **free()**, você não deverá mais utilizar o ponteiro utilizado nessa chamada para acessar o espaço liberado; caso contrário, seu programa poderá apresentar um comportamento indefinido.

É importante salientar que, apesar de um ponteiro ser considerado inválido após ser utilizado numa chamada da função **free()**, não é possível detectar essa situação, pois ele continua apontando para o mesmo endereço do bloco liberado. Portanto sugere-se que sempre se atribua **NULL** a um ponteiro logo após ele ser utilizado numa chamada da função **free()**.

O parâmetro passado numa chamada da função **free()** deve ser **NULL** ou um ponteiro que esteja correntemente apontando para o início de um bloco alocado com alguma das funções de alocação descritas aqui. Esse ponteiro também não deve ter sido previamente liberado ou passado como parâmetro para **realloc()**. Se essas recomendações não forem seguidas numa chamada de **free()**, o resultado da chamada será imprevisível e o programa que a contém poderá ser abortado ou apresentar um comportamento errático.

12.2.4 realloc()

A função **realloc()**, tipicamente utilizada para redimensionar blocos previamente alocados dinamicamente, possui dois parâmetros. O primeiro parâmetro deve ser um ponteiro para o início de um bloco de memória alocado utilizando **malloc()**, **calloc()** ou a própria função **realloc()** e o segundo parâmetro especifica um novo tamanho desejado para o bloco. O protótipo de **realloc()** é:

```
void *realloc(void *ptr, size_t tamanho)
```

A Figura 12–1 e a Figura 12–2 ilustram o funcionamento de **realloc()**. Se o novo tamanho, especificado pelo segundo parâmetro de **realloc()**, for menor ou maior do que o tamanho atual do bloco apontado pelo primeiro parâmetro, essa função tentará alocar um bloco com o tamanho especificado. Então, os bytes do bloco atual serão copiados para o novo bloco até o limite do menor dos dois blocos. Quer dizer, se o novo tamanho for menor do que o tamanho do bloco atual, apenas os bytes iniciais do bloco atual que couberem no novo bloco serão copiados (v. Figura 12–1), enquanto, se o novo tamanho for maior do que o tamanho atual do bloco, todo o conteúdo do bloco atual será copiado para o início do novo bloco. Nesse último caso, os bytes restantes terão valores indeterminados (v. Figura 12–2).

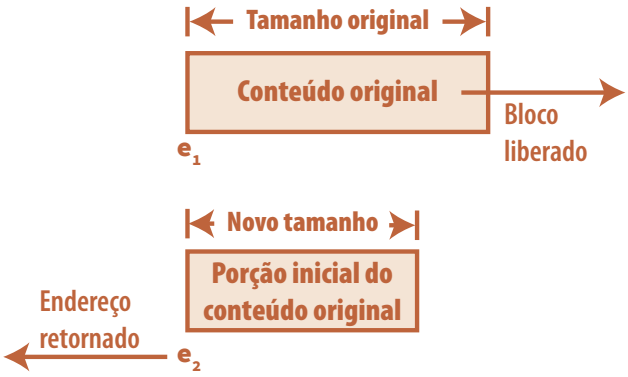

FIGURA 12–1: FUNÇÃO REALLOC(): NOVO BLOCO É MENOR DO QUE O BLOCO ORIGINAL

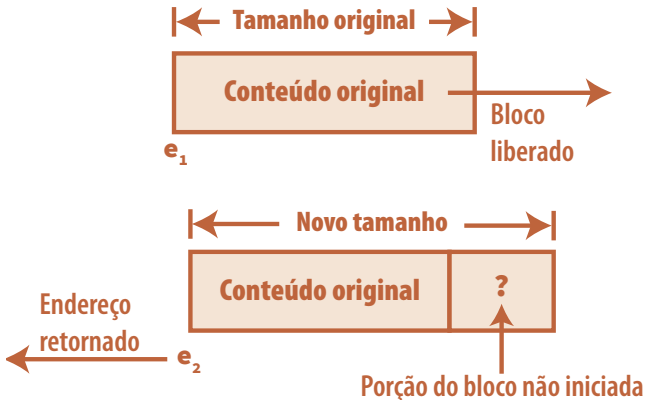

FIGURA 12–2: FUNÇÃO REALLOC(): NOVO BLOCO É MAIOR DO QUE O BLOCO ORIGINAL

Em qualquer chamada de **realloc()**, um dos seguintes valores pode ser retornado:

- **NULL**. Nesse caso, o bloco apontado pelo primeiro parâmetro da função permanece intacto, o que significa que nem esse bloco foi realocado em novo endereço nem seu tamanho foi alterado. Em outras palavras, a chamada de **realloc()** foi absolutamente ineficaz. Deve-se ressaltar que, nessa situação, o segundo parâmetro continua apontando para um bloco válido.
- Um **endereço válido**. Nesse caso, o tamanho do bloco foi alterado e ele pode ter sido realocado em nova posição. Assim, o ponteiro usado como primeiro parâmetro deve ser considerado inválido.

O uso recomendado de **realloc()** requer que o retorno dessa função seja atribuído a um ponteiro diferente daquele passado como primeiro parâmetro para ela, como mostra o seguinte fragmento de programa:

```
int *pNovoBloco, *pBloco = malloc(10*sizeof(int));
...
pNovoBloco = realloc(pBloco, 20*sizeof(int));

if (pNovoBloco) {
   pBloco = pNovoBloco;
}
```

Ao final da execução desse trecho de programa, o ponteiro `pBloco` poderia continuar sendo utilizado, quer a solicitação de redimensionamento do bloco fosse atendida ou não. Mas, se o programador não seguir a recomendação acima e escrever:

```
int *pBloco = calloc(10, sizeof(int));
...
pBloco = realloc(pBloco, 20*sizeof(int));
```

Se essa última chamada de **realloc()** retornar **NULL**, o bloco para o qual o ponteiro `pBloco` apontava estará irremediavelmente perdido, pois seu único meio de acesso (i.e., o próprio ponteiro `pBloco`) deixará de apontar para o bloco. Portanto siga sempre o conselho preconizado no quadro a seguir:

Recomendação ***Nunca atribua o retorno de realloc() ao mesmo ponteiro usado como primeiro parâmetro dessa função.***

Se **NULL** for passado como primeiro parâmetro para a função **realloc()**, ela se comportará como **malloc()** e tentará alocar um bloco com o tamanho especificado pelo segundo parâmetro. Se o segundo parâmetro for igual a zero e o primeiro parâmetro não for **NULL** numa chamada de **realloc()**, essa função se comportará como **free()** (i.e., ela liberará o espaço em memória apontado pelo ponteiro). Contudo, essas duas últimas formas de utilização de **realloc()** são atípicas e não há razão para empregá-las.

A Seção 12.6.1 apresenta um exemplo prático de uso de **realloc()**.

12.3 Ponteiros Genéricos e o Tipo void *

Recorde-se que dois ponteiros são compatíveis apenas quando eles são exatamente do mesmo tipo (v. Seção 5.2). Entretanto, existem ponteiros, denominados **ponteiros genéricos**, que são compatíveis com ponteiros de quaisquer tipos. Sintaticamente, um ponteiro genérico é um ponteiro do tipo **void** *. Por exemplo, o ponteiro `ponteiroGenerico` abaixo é um ponteiro genérico:

```
void *ponteiroGenerico;
```

O tipo **void** * é normalmente utilizado em duas situações:

- Como tipo de **retorno de função** [p. ex., **malloc()**]
- Como tipo de **parâmetro de função** [p. ex., **free()**]

No primeiro caso, o endereço retornado pela função pode ser implicitamente convertido em qualquer tipo de ponteiro. Por exemplo, as funções de alocação de memória **malloc()**, **calloc()** e **realloc()** têm tipo de retorno **void** * e isso significa que os endereços retornados por essas funções podem ser atribuídos a ponteiros de quaisquer tipos sem a necessidade de conversão explícita, conforme já foi visto.

No segundo caso de uso de **void** *, esse tipo é utilizado para representar parâmetros compatíveis com qualquer tipo de ponteiro. Por exemplo, o parâmetro formal da função **free()** (v. Seção 12.2.3) é do tipo **void** *, o que permite a passagem de parâmetros reais de quaisquer tipos de ponteiros, sem necessidade de conversão explícita.

Quando uma variável ou, mais comumente, um parâmetro é do tipo **void** *, é necessário convertê-lo para um tipo de ponteiro conhecido pelo compilador antes que seja permitida a aplicação do operador de indireção sobre ele. Por exemplo, se você tentar compilar o seguinte programa, obterá duas mensagens de erro relacionadas às duas primeiras chamadas de **printf()**.

```
#include <stdio.h>

typedef enum {INTEIRO, REAL} tTipoDeDado;

void ImprimeValor(void *valor, tTipoDeDado tipo)
{
   if (tipo == INTEIRO) {
      printf("Valor: %d\n", *valor); /* ILEGAL */
   } else if (tipo == REAL) {
      printf("Valor: %f\n", *valor); /* ILEGAL */
   } else {
      printf("Tipo desconhecido\n");
   }
}

int main(void)
{
   int i = 5;

   ImprimeValor(&i, INTEIRO);

   return 0;
}
```

Os erros nesse programa dizem respeito às duas aplicações do operador de indireção sobre ponteiros genéricos (i.e., `*valor`) nas duas primeiras chamadas de **printf()**. Quer dizer, para que o compilador seja capaz de interpretar o conteúdo para o qual o ponteiro `valor` aponta, ele precisa conhecer o tipo desse conteúdo. Mas, como o referido ponteiro é genérico, o conteúdo para o qual ele aponta pode ser de qualquer tipo. A solução para esse impasse é converter explicitamente os ponteiros para os tipos desejados por meio dos operadores **(int *)**, no primeiro caso, e **(double *)**, no segundo caso. Assim, as duas chamadas incorretas de **printf()** no programa acima podem ser corrigidas reescrevendo-as como:

```
printf("Valor: %d\n", *(int *)valor);
```

e

```
printf("Valor: %f\n", *(double *)valor);
```

Pela mesma razão exposta acima, não é permita nenhuma operação de aritmética de ponteiros (v. Seção 8.6) sobre um ponteiro genérico que não tenha sido convertido explicitamente para um tipo de ponteiro conhecido.

12.4 Espaço de Execução de um Programa

Tipicamente, o espaço reservado em memória para a execução de um programa é dividido em quatro partições como mostra a Figura 12–3.

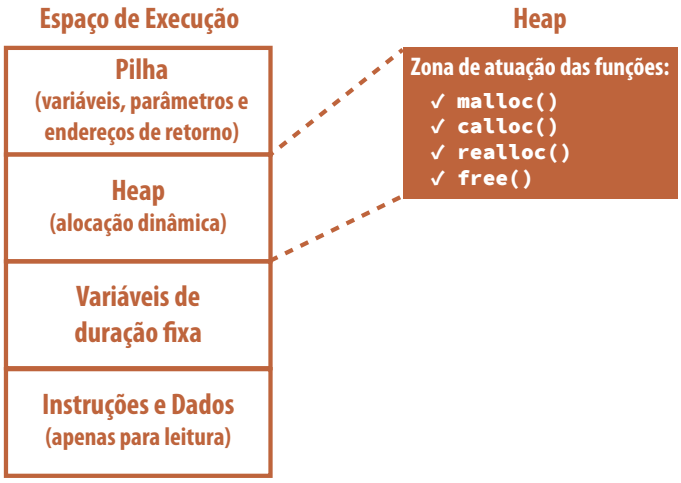

Figura 12–3: Espaço de Execução de um Programa

As partições de memória reservadas para a execução de um programa podem ser descritas de modo bem simplificado como:

- **Instruções e dados**. A partição na parte inferior da **Figura 12–3** é reservada para conter as instruções do programa em linguagem de máquina bem como os dados do programa que não devem ser alterados. Isto é, esse espaço é considerado apenas para leitura e muitos sistemas operacionais encerram um programa se ele tentar alterar o conteúdo dessa área. O espaço alocado para essa partição tem tamanho fixo durante toda a execução do programa.
- **Variáveis de duração fixa**. A segunda partição de baixo para cima na **Figura 12–3** abriga variáveis de duração fixa e o espaço alocado é fixo durante toda a execução do programa.
- **Heap**. A terceira partição de baixo para cima na **Figura 12–3** é reservada para alocação dinâmica de memória e o espaço alocado nessa partição aumenta ou diminui de tamanho de acordo com os blocos alocados e liberados dinamicamente durante a execução do programa. Essa partição será discutida em profundidade mais adiante nesta seção.
- **Pilha** (*stack*, em inglês). A partição no topo da figura é denominada *pilha* porque seu funcionamento se assemelha ao de uma pilha de objetos. Isto é, os blocos armazenados nesse espaço são liberados na ordem inversa de alocação (como ocorre, por exemplo, com uma pilha de pratos). Alocação na pilha ocorre quando uma função é chamada e liberação ocorre quando uma função retorna. Ou seja, quando uma função é chamada, nesse espaço são alocados os parâmetros da função, suas variáveis de duração automática e o endereço da instrução que será executada quando a função retornar. Mais precisamente, a pilha de execução de um programa é dividida em blocos contíguos em memória denominados *stack frames*. A cada chamada de função, é criado um *stack frame* para essa chamada contendo: o endereço da instrução que fez a chamada, cópias dos parâmetros reais utilizados na chamada e as variáveis locais de duração automática da função. Quando a função retorna, o espaço alocado em memória para o *stack frame* da chamada é liberado. Em qualquer instante, a pilha de execução contém todos os *stack frames* associados a funções correntemente em execução (i.e., que ainda não retornaram).

Às vezes, uma função de alocação dinâmica deixa de alocar um bloco em memória em virtude de **fragmentação de heap**, que ocorre em consequência de várias alocações e liberações de blocos de tamanhos variados durante a execução do programa. Pode-se fazer uma analogia entre fragmentação de *heap* e a fragmentação que frequentemente ocorre em meios de armazenamento não volátil, notadamente em discos rígidos.

De modo análogo à causa de fragmentação de *heap*, a fragmentação de um disco rígido é causada por criação e remoção frequentes de arquivos de tamanhos diferentes. Entretanto, diferentemente do que ocorre em sistemas de arquivos, nos quais as partes que compõem um arquivo não precisam ser contíguas, os bytes que compõem um bloco em memória principal devem ser contíguos. Assim, quando ocorre fragmentação de *heap*, pode ser impossível alocar um bloco de memória contíguo, mesmo que a quantidade total de memória disponível no *heap* seja maior do que o tamanho do bloco requisitado.

O programa apresentado a seguir é usado para medir a capacidade disponível do heap usado pelo próprio programa em megabytes.

```
#include <stdio.h>
#include <stdlib.h>

#define MEGABYTE 1048576

int main(void)
{
   int MB = 0; /* Conta o número de megabytes alocados */

   while (malloc(MEGABYTE))
      ++MB;

   printf("\nForam alocados %d MB\n", MB);

   return 0;
}
```

Esse programa aloca blocos de um megabyte sucessivamente até esgotar a capacidade do heap. O número de blocos alocados é contado com o objetivo de determinar a capacidade aproximada do heap em megabytes. Contudo, para a medição ser precisa, dever-se-iam alocar blocos de um byte, mas, assim, o programa levaria um tempo considerável para esgotar o heap.

Quando compilado com GCC e executado em Windows XP, o programa apresenta o seguinte resultado na tela:

```
Foram alocados 1919 MB
```

É importante emparelhar cada chamada de **malloc()**, **calloc()** ou **realloc()** com uma correspondente chamada de **free()**. Caso contrário, provavelmente, seu programa apresentará um problema conhecido como **escoamento de memória**. O último programa apresentado simula escoamento de memória. Nele, a cada chamada de **malloc()**, o endereço do bloco mais recentemente alocado é sobrescrito e jamais poderá ser acessado ou liberado. Infelizmente, na prática, escoamento de memória não é tão evidente assim.

12.5 Testando Alocação Dinâmica

Todas as funções de alocação dinâmica de memória retornam **NULL** quando não é possível alocar um bloco requerido em virtude de esgotamento ou fragmentação de *heap*. Portanto é sempre importante testar o valor retornado por essas funções antes de tentar utilizá-lo para acessar um bloco que não se tem certeza se foi realmente alocado. Caso o valor retornado por uma função de alocação seja **NULL**, o programador deve tomar as devidas providências antes de prosseguir. Nesse caso, talvez, o programa precise ser abortado graciosamente (v. **Seção 6.10**), se o bloco requisitado for crucial para o prosseguimento do programa. Por exemplo, o fragmento de programa a seguir mostra como o programador deve proceder após uma chamada de **malloc()** [ou **calloc()** ou **realloc()**]:

```
ptrAluno = malloc(sizeof(tAluno));

if  (ptrAluno != NULL){ /* Bloco foi alocado */
   /* Aqui, o bloco foi alocado e seu conteúdo pode ser acessado com segurança */
} else { /* Bloco NÃO foi alocado */
   /* Aqui o programador deve tomar as providências cabíveis quando não */
   /* é possível alocar o  espaço desejado. Talvez seja preciso abortar */
   /* o programa, mas pode ser que haja alternativa menos drástica,     */
   /* dependendo da situação.                                           */
}
```

O teste:

```
if (ptrAluno != NULL)
```

pode ser escrito, de forma equivalente, como:

```
if (ptrAluno)
```

Essa última forma é a preferida pela maioria dos programadores de C.

12.6 Exemplos de Programação

12.6.1 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 (representado por `'\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 de (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 o 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 12–2 a comparação entre essa função e a função **fgets()**, discutida na Seção 11.9.6.

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 parâmetro	O array no qual os caracteres são armazenados é alocado dinamicamente
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 12–2: 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 essa linha 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 redimensionamento do array.
  - 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 em discussão. Esse laço, deve-se frisar, não tem condição natural de parada, pois seu encerramento acontecerá no corpo dele. 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;
}
```

  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 for 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()**. O objetivo dessa chamada é acrescentar um número de bytes igual à constante simbólica `TAMANHO_BLOCO` ao tamanho do array. Também, 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 perdido (v. adiante). Aparentemente, uma boa ideia seria aumentar o tamanho do array a cada caractere lido, pois, assim, não haveria nenhum desperdício de memória. Mas, de fato, essa não é uma boa alternativa por causa do ônus associado a cada chamada de **realloc()** (v. Seção 12.2.4).

A próxima instrução no corpo do laço testa se a 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 `LeLinhaIlimitada()` não foi bem sucedida.

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

Ainda nesse caso, 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 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;
```

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 em discussão retorna **NULL**.

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

Essa instrução **if** inclui 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 esse parâmetro, 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, nessa situação, é bem mais sensato retornar `ar` do que **NULL**.

Solução de (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*** Conteudo do Arquivo %s ***\n", NOME_ARQ);

      /* 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 AnedotaBulgara.txt ***
>>> 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
```

12.6.2 Jogo de Palavras (com Ajuda dos Universitários)

Problema: (a) Escreva uma função que embaralha strings, que, supostamente, representam palavras. (b) Escreva um programa que lê uma palavra de um arquivo de texto contendo uma palavra por linha, embaralha a palavra usando a função solicitada no item (a), apresenta a palavra embaralhada para o usuário e desafia-o a adivinhar qual é a palavra. O número de chutes permitidos deve ser previamente especificado por uma constante simbólica e o programa deve ainda oferecer três níveis de ajuda para facilitar a adivinhação: (1) qual é a primeira letra da palavra (2) qual é a última letra da palavra e (3) qual é a segunda letra da palavra. A cada ajuda recebida, o usuário perde direito a um chute.

Solução de (a):

```
/****
 * EmbaralhaPalavra(): Embaralha os caracteres de um string
 *
 * Parâmetros:
 *      embaralho (saída) - o string embaralhado
 *      palavra (entrada) - string que será embaralhado
 *
 * Retorno: Endereço do string embaralhado, se não ocorrer erro. NULL, se ocorrer erro
 *
 * Observação: Esta função não verifica se o string contém apenas
 *             letras. Portanto ela serve para embaralhar qualquer tipo de string.
 ****/
char *EmbaralhaPalavra(char *embaralho, const char *palavra)
{
   int   i, j, tam;
   char *p;
```

```
   tam = strlen(palavra); /* Calcula o tamanho da palavra */

      /* Aloca um array auxiliar. Acrescenta-se 1 ao tamanho da palavra */
      /* para levar em conta o caractere terminal de string             */
   p = malloc(tam + 1);

      /* Se o array não foi alocado, não é possível continuar */
   if (!p) {
      return NULL; /* Game over! */
   }

      /* Copia o string que será embaralhado para o array auxiliar */
   strcpy(p, palavra);

      /* Embaralha o string trocando os caracteres de posição */
   for (i = 0; i < tam; ++i) {
      while (1) {
            /* Sorteia o caractere do string apontado por p que será     */
            /* armazenado na posição i do array apontado por 'embaralho' */
         j = rand()%tam;

         /*************************************************************************/
         /* Caracteres que já foram usados são substituídos por '\0' no array     */
         /* p[]. A instrução if verifica se o caractere sorteado já foi usado.    */
         /* Se esse não for o caso, armazena-se o caractere no array embaralho[], */
         /* substitui-se o caractere usado no array p[] por '\0' e passa-se para  */
         /* o próximo caractere do array embaralho[]. Se o caractere já tiver     */
         /* sido usado, volta-se ao início do laço while e faz-se novo sorteio.   */
         /*************************************************************************/

         if (p[j] != '\0') {
              /* O caractere de índice j do string apontado por */
              /* p ainda não foi usado no embaralho. Então, ele */
              /* é armazenado no array que contém  o embaralho  */
            embaralho[i] = p[j];

               /* Agora, o caractere de índice j já foi usado   */
               /* no  embaralho. Assim, coloca-se um caractere  */
               /* terminal nessa posição para indicar esse fato.*/
            p[j] = '\0';

            break; /* Encerra o laço while */
         }
      } /* while */
   } /* for */

      /* Armazena o caractere terminal no array      */
      /* embaralho[] para que ele contenha um string */
   embaralho[tam] = '\0';

      /* Libera espaço ocupado pelo array auxiliar, pois ele não é mais necessário */
   free(p);

   return embaralho;
}
```

Análise: A abordagem de embaralho de string utilizada pela função `EmbaralhaPalavra()` é a seguinte:

1. Um array auxiliar com tamanho suficiente para armazenar a palavra (string) a ser embaralhada é criado dinamicamente.
2. O conteúdo da palavra é copiado para esse array para que o string original não seja alterado por causa da estratégia de embaralho escolhida (v. próximo passo).

3. O índice do caractere na palavra que será armazenado no elemento de índice `i` do array apontado por `embaralho` é sorteado no corpo do laço **for** da função. Se o índice sorteado coincidir com um caractere que já tenha sido sorteado antes, faz-se um novo sorteio. Para indicar que um determinado caractere da palavra já foi sorteado, substitui-se esse caractere por um caractere nulo. Isso justifica o uso do array auxiliar para armazenar a palavra que será embaralhada, pois, caso contrário, ela estaria irremediavelmente perdida ao final do processo.

Evidentemente, existem outras abordagens de embaralho de strings além daquela usada pela função `EmbaralhaPalavra()`. Mas, provavelmente, a abordagem escolhida é a mais fácil de entender (e este é um livro didático...).

Solução de (b):

```
/****
 *
 * main(): Lê uma palavra de um arquivo contendo uma palavra por
 *         linha, embaralha a palavra, apresenta a palavra
 *         embaralhada  ao usuário e desafia-o a adivinhar qual é a palavra.
 *
 * Parâmetros: Nenhum
 *
 * Retorno: 0, se não ocorrer nenhum erro; 1, em caso contrário
 *
 ****/
int main(void)
{
   int   i,
         nPalavras, /* Número de palavras no arquivo */
         indicePalavra, /* Índice da palavra sorteada */
         tamanho, /* Tamanho da palavra sorteada no arquivo */
         nAjuda = 0; /* Número de auxílios oferecidos ao usuário */
   char *palavra, /* Aponta para o array contendo a palavra sorteada */
        *embaralho, /* Aponta para o array contendo a palavra embaralhada */
        *chute; /* Ponteiro para o array que conterá cada tentativa do usuário */
   FILE *stream; /* Stream associado ao arquivo que contém o banco de palavras */
      /* Apresenta o programa */
   printf("\n\t>>> Este programa embaralha uma palavra e da' "
          "chances\n\t>>> para voce adivinhar qual e' a palavra "
          "original.\n\t>>> Digite '?' para solicitar ajuda aos "
          "universitarios\n\t>>> (3 vezes apenas), mas "
          "voce perde um chute a cada ajuda\n\n");

      /* Tenta abrir o arquivo contendo as palavras para leitura em modo texto */
   stream = fopen(ARQ_PALAVRAS, "r");

      /* Se o arquivo não foi aberto, o jogo não pode continuar */
   if (!stream) {
      printf( "\nNao foi possivel abrir o arquivo %s\n"
              "e o jogo sera' adiado\n", ARQ_PALAVRAS );
      return 1; /* Game over! */
   }

      /* Determina o número de palavras do arquivo, */
      /* que contém uma palavra por linha           */
   nPalavras = NumeroDeLinhas(stream);

   srand(time(NULL)); /* Inicia o gerador de números aleatórios */
```

```
   /* Sorteia o índice da palavra no arquivo. A indexação começa em 1. */
indicePalavra = rand()%nPalavras + 1;

   /* Assegura que a leitura começa no início do arquivo */
rewind(stream);

   /* Lê e descarta cada palavra até chegar à palavra sorteada */
for (i = 1; i < indicePalavra; ++i) {
      /* A função LeLinhaIlimitada() aloca espaço dinamicamente. Portanto */
      /* é necessário liberar o espaço ocupado por cada linha lida.       */
   free(LeLinhaIlimitada(NULL, stream));
}

   /* Lê no arquivo a palavra sorteada e armazena seu */
   /* número de caracteres na variável 'tamanho'      */
palavra = LeLinhaIlimitada(&tamanho, stream);

   /* O arquivo não é mais necessário */
FechaArquivo(stream, ARQ_PALAVRAS);

   /* Aloca espaço para conter a palavra embaralhada. */
   /* É necessário acrescentar 1 porque o tamanho da  */
   /* palavra não inclui o caractere terminal '\0'.   */
embaralho = malloc(tamanho + 1);

   /* Tenta embaralhar a palavra. Se não for possível, encerra o programa.  */
if ( !EmbaralhaPalavra(embaralho, palavra) ) {
   printf("\nOcorreu um erro no embaralho\n");
   return 1;
}

   /* Desafia usuário apresentando-lhe a palavra embaralhada */
printf( "\nVoce tem %d chances para adivinhar que "
        "palavra e' esta: %s\n", CHANCES, embaralho );

   /* Inicia 'chute' com NULL para não haver problema */
   /* na primeira chamada de free() no corpo do laço  */
chute = NULL;

   /* Lê cada chute do usuário e verifica se ele acertou qual é a palavra certa */
for (i = 1; i <= CHANCES; ++i) {
      /* Libera o espaço ocupado pelo último chute do        */
      /* usuário. Não haverá problema se ainda não houve     */
      /* chute pois a variável 'chute' foi iniciada com NULL */
   free(chute);

      /* Lê um chute do usuário, testando se ocorre erro de leitura */
   printf("\n\t>>> %do. chute: ", i);
   if (!(chute = LeLinhaIlimitada(NULL, stdin))) {
      printf("\nOcorreu um erro de leitura\n");
      return 1; /* Se a função LeLinhaIlimitada() retornou NULL, deve */
                /* ter ocorrido erro. Talvez o usuário tenha digitado */
                /* ^D (Unix) ou ^Z (Windows/DOS).                     */
   }

      /* Se o primeiro caractere do chute do usuário */
      /* for '?', interpreta-se que ele deseja ajuda */
   if (*chute == '?') {
      if (!AjudaUniversitarios(palavra, ++nAjuda)) {
            /* Se ocorreu ajuda, o número de chutes é decrementado */
         --i;
      }
```

```
            /* Salta o resto do laço pois não houve chute */
            continue;
        }

            /* Verifica se o chute do usuário foi correto. O chute */
            /* precisa ser convertido em letras maiúsculas porque  */
            /* as palavras do arquivo são assim.                   */
        if (!strcmp(ConverteEmMaiusculas(chute), palavra)) {
            printf("\a\n\t>>> Parabens. Voce acertou. <<<\n");
            return 0; /* Se o usuário acertou, o jogo acaba */
        } else if (i < CHANCES) {
            printf("\nErrou. Tente novamente.\n");
        }
    }
        /* Se não houve retorno no corpo do último laço for, o usuário não acertou */
    printf( "\n\t>>> Infelizmente, voce nao acertou."
            "\n\t>>> A palavra era: \"%s\"\n", palavra );

        /* Libera os espaços ocupados pela palavra e pelo embaralho. */
        /* Isso não é realmente necessário, já que o programa        */
        /* encerrará em seguida. Mas, preserva o bom hábito.         */
    free(palavra);
    free(embaralho);

    return 0;
}
```

Análise: Após abrir o arquivo contendo as palavras do jogo, a função **main()** chama a função `NumeroDeLinhas()` (v. Seção 11.15.5) para contar o número de linhas desse arquivo, que coincide com o número de palavras, já que cada linha do arquivo contém exatamente uma palavra. Então, após iniciar o gerador de números aleatórios, o índice da linha do arquivo que contém a palavra é sorteado. Em seguida, com auxílio da função `LeLinhaIlimitada()` (v. Seção 12.6.1), a palavra sorteada é lida no arquivo e embaralhada pela função `EmbaralhaPalavra()`, apresentada como solução do item (a) do problema. O restante da função **main()** é dedicado à interação com usuário. Quer dizer, a palavra embaralhada é apresentada e o programa espera que o usuário acerte qual é a palavra correta ou suas chances estejam esgotadas. A função **main()** chama ainda as seguintes funções:

- `ConverteEmMaiusculas()` (v. Seção 9.7.2) para converter os chutes dos usuários em letras maiúsculas, já que as palavras do arquivo do qual as palavras do jogo se originam estão todas em letras maiúsculas[2].
- `FechaArquivo()` (v. Seção 11.5) para fechar o arquivo que contém as palavras quando ele deixa de ser necessário.
- `AjudaUniversitarios()` que oferece os três níveis de ajuda requeridos na definição do problema. A implementação dessa função é apresentada a seguir[3]:

```
/****
 * AjudaUniversitarios(): Oferece ajuda ao jogador
 *
 * Parâmetros: palavra (entrada) - a palavra para a qual será apresentada ajuda
 *             n (entrada) - índice da ajuda
 *
 * Retorno: 1, quando é concedida ajuda. 0, em caso contrário.
 ****/
```

[2] A lista de palavras utilizada por esse programa e que pode ser encontrada no site do livro, foi criada e gentilmente cedida por Valdir Jorge, que é analista de sistemas e programador na Universidade Concórdia, em Montreal, Canadá. O autor deste livro penhoradamente agradece a gentileza.

[3] A expressão *ajuda dos universitários* é usada aqui como uma paródia de um chavão derivado do programa *Show do Milhão* da rede de TV SBT.

```
int AjudaUniversitarios(const char *palavra, int n)
{
   switch (n) { /* Seleciona o nível de ajuda */
      case 1: /* Primeiro nível de ajuda */
         printf("\n\t>>> A primeira letra e': %c\n", *palavra);
         return 1; /* Uma ajuda foi concedida */
      case 2: /* Segundo nível de ajuda */
         printf( "\n\t>>> A ultima letra e': %c\n", *(strchr(palavra, '\0') - 1) );
         return 1; /* Outra ajuda foi concedida */
      case 3: /* Terceiro nível de ajuda */
         printf( "\n\t>>> (Ultima ajuda) A segunda letra e': %c\n", palavra[1] );
         return 1; /* Mais outra ajuda foi concedida */
      default: /* Nem em Silvio Santos há mais ajuda */
         printf("\n\t>>> Nao ha mais ajuda dos universitarios\n");
         return 0; /* Não houve ajuda */
   }
}
```

Análise: A função `AjudaUniversitarios()` oferece três níveis de ajuda:

[1] Na primeira chamada dessa função (i.e., quando o parâmetro `n` é igual a `1`), ela informa qual é a primeira letra da palavra que deve ser adivinhada.

[2] Na segunda chamada da função, ela informa qual é a última letra da palavra.

[3] A terceira chamada da função informa qual é segunda letra da palavra.

Em chamadas subsequentes (i.e., quando `n` é maior do que `3`), a função informa que não há mais ajuda disponível.

Complemento do programa: Para completar o programa inclua as seguintes linhas no início do arquivo-fonte:

```
/********************** Includes *************************/

#include <stdio.h>   /* Entrada e saída             */
#include <stdlib.h>  /* Alocação dinâmica           */
#include <time.h>    /* Função time()               */
#include <string.h>  /* Processamento de strings    */
#include <ctype.h>   /* Classificação de caracteres */
#include "leitura.h" /* LeituraFacil                */

/***************** Constantes Simbólicas ******************/

   /* Nome do arquivo que contém as palavras */
#define ARQ_PALAVRAS "ListaDePalavras.txt"

#define MAX_PALAVRA  50 /* Tamanho máximo de uma palavra */
#define CHANCES       5  /* Número máximo de chances que  */
                         /* o usuário terá para adivinhar */
#define TAMANHO_BLOCO 256 /* Tamanho do acréscimo do bloco usado na */
                          /* leitura a cada chamada de realloc()    */

/************************ Alusões ************************/

extern void FechaArquivo(FILE *stream, const char *nomeArq);
extern int NumeroDeLinhas(FILE*  stream);
extern char *LeLinhaIlimitada(int *tam, FILE *stream);
extern char* ConverteEmMaiusculas(char *str);
extern char *EmbaralhaPalavra( char *embaralho, const char *palavra );
extern int AjudaUniversitarios(const char *palavra, int n);
```

Exemplo de execução do programa:

```
        >>> Este programa embaralha uma palavra e da' chances
        >>> para voce adivinhar qual e' a palavra original.
        >>> Digite '?' para solicitar ajuda aos universitarios
        >>> (3 vezes apenas), mas voce perde um chute a cada ajuda
Voce tem 5 chances para adivinhar que palavra e' esta: IEODDTNRSU

        >>> 1o. chute: ?

        >>> A primeira letra e': D

        >>> 2o. chute: ?

        >>> A ultima letra e': O

        >>> 3o. chute: ?

        >>> (Ultima ajuda) A segunda letra e': E

        >>> 4o. chute: ?

        >>> Nao ha' mais ajuda dos universitarios

        >>> 4o. chute: desmentido

Errou. Tente novamente.

        >>> 5o. chute: ?

        >>> Nao ha' mais ajuda dos universitarios

        >>> 5o. chute: desisto

        >>> Infelizmente, voce nao acertou.
        >>> A palavra era: "DESNUTRIDO"
```

12.6.3 A Urupema de Eratóstenes

Preâmbulo: A **Urupema de Eratóstenes**[4] é uma técnica utilizada para determinar os números primos menores do que um determinado valor por meio da exclusão dos números que não são primos no intervalo constituído pelo menor número primo (i.e., 2) e o valor supracitado. Isto é, os números que são múltiplos de algum primo são assinalados como compostos (i.e., não primos), de modo que, ao final do processo, os valores que não forem assinalados são todos primos.

Problema: Escreva uma função que encontra todos os números primos menores do que um valor especificado como parâmetro e exibe-os na tela usando a Urupema de Eratóstenes. (b) Escreva um programa que solicita um valor inteiro positivo ao usuário e apresenta os números primos menores do que ou iguais ao valor introduzido usando a função solicitada no item (a).

Solução de (a):

```
/****
 * Eratostenes(): Encontra todos os números primos menores do que
 *                o valor especificado como parâmetro e exibe-os
 *                na tela usando a técnica do Crivo de Eratóstenes
 *
 * Parâmetros: n (entrada) - maior valor que se verificará se é primo
 *
 * Retorno: 1, se ocorrer erro. 0, se não ocorrer erro.
 ****/
int Eratostenes(int n)
```

[4] O nome mais conhecido dessa técnica em português é *Crivo de Eratóstenes*, mas *crivo*, *peneira*, *joeira*, *coador* ou... *urupema* também serve como rótulo inicial dela, pois seu nome é derivado do fato de números primos serem separados (i.e., *peneirados*) daqueles que não são primos.

```
{
   int *ar,      /* Array usado como peneira */
       primo,    /* Armazena um número primo */
       tamanho, /* Tamanho da peneira       */
       i, j;

      /* Verifica se valor recebido como parâmetro é inconveniente */
   if (n <= 0) {
      return 1; /* Valor deveria ser pelo menos igual a 1 */
   }

      /* O menor número primo é 2. Logo, a peneira deverá armazenar os números */
      /* 2, 3, ..., n. Assim, o tamanho da peneira (array) deve ser n - 1.     */
   tamanho = n - 1;

      /* Tenta alocar o array que servirá de coador */
   ar = calloc(tamanho, sizeof(int));

      /* Se não houve alocação, não é possível prosseguir */
   if (!ar) {
      return 1; /* Array não foi alocado */
   }

      /* Inicia os elementos do array com valores entre 2 e n */
   for(i = 0; i < tamanho; i++) {
      ar[i] = i + 2; /* 2 é o menor primo */
   }

      /* Seleciona um número primo a partir do primeiro elemento do array e       */
      /* atribui zero a todos os demais elementos do array que são seus múltiplos */
   for(i = 0; i < tamanho; i++) {
         /* O próximo primo considerado é o elemento  */
         /* imediato do array que é diferente de zero */
      if(ar[i]) {
         primo = ar[i]; /* Se não é zero, é primo */
      } else {
            /* O restante do laço deve ser saltado, pois */
            /* não foi encontrado um novo número primo   */
         continue;
      }

            /* Atribui zero a cada elemento do array que */
            /* é múltiplo do último primo encontrado     */

         /* O primeiro múltiplo de um número primo é seu dobro */
      j = 2*primo;

      while(j <= n) {
            /* Atribui zero ao elemento que é igual ao último múltiplo */
            /* de 'primo'. Note que o elemento cujo valor é j está     */
            /* armazenado no elemento de  índice j - 2.                */
         ar[j - 2] = 0;

         j = j + primo; /* Obtém o próximo múltiplo */
      } /* while */
   } /* for */

      /* Neste ponto, todos os elementos do array que não */
      /* são primos têm valor zero; i.e., os elementos do */
      /* array foram joeirados na Joeira de Eratóstenes   */

   printf("\n\t  *** Primos entre 2 e %d ***\n", n);
```

```
      /* Apresenta na tela os elementos que não  */
      /* são nulos; i.e., aqueles que são primos */
   for(i = 0, j = 0; i < tamanho; ++i) {
      if(ar[i]) { /* Se não é zero, é primo */
            /* Quebra linha quando o número de primos */
            /* exibidos atinge um valor especificado  */
         if (!(j%PRIMOS_POR_LINHA)) {
            printf("\n");
         }

         printf("%4d  ", ar[i]); /* Exibe mais um número primo */

         ++j; /* Mais um primo foi exibido */
      }
   }

   putchar('\n'); /* Embelezamento */
   free(ar); /* Libera espaço ocupado pelo array */

   return 0; /* A urupema funcionou */
}
```

Análise: Brincadeiras à parte e sem levar em consideração as instruções que testam condições de exceção, o funcionamento da função `Eratostenes()` é o seguinte:

- O tamanho do array que servirá como peneira é calculado como:

  ```
  tamanho = n - 1;
  ```

  Nessa instrução, `n` é o parâmetro único da função e representa o maior número que será testado se é primo ou não. Subtrai-se `1` do tamanho do array porque os valores que serão testados são `2`, `3`, `...`, `n`; portanto, existem `n - 1` números a serem testados (lembre-se que o primeiro número primo é `2`).

- Em seguida, a função aloca espaço para o referido array por meio da chamada de **calloc**():

  ```
  ar = calloc(tamanho, sizeof(int));
  ```

- No primeiro laço **for** da função em discussão, os valores que serão testados são armazenados no array:

  ```
  for (i = 0; i < tamanho; i++) {
     ar[i] = i + 2;
  }
  ```

  Do modo como os elementos do array são iniciados, o primeiro elemento é `2`, o segundo elemento é `3` e assim por diante.

- O segundo laço **for** é efetivamente responsável por peneirar os números, separando-os em primos e compostos. Essa separação ocorre atribuindo-se zero aos elementos que não são primos. Esse laço funciona do seguinte modo:

  1. Seleciona-se o próximo número primo no array (i.e., o próximo elemento do array que não é igual a zero), sendo que essa seleção começa com o primeiro elemento do array (i.e., `2`), que é primo.
  2. Atribui-se zero a cada elemento do array que é múltiplo do número primo selecionado no passo anterior. Aqui, leva-se em conta que o primeiro múltiplo de um número primo é o seu dobro e os múltiplos seguintes são obtidos somando-se o múltiplo anterior com o próprio número.

O restante da função em discussão é dedicada à exibição dos números primos que foram obtidos por meio do processo descrito. Esse trecho da função é relativamente trivial e não requer comentários adicionais.

Solução de (b):

```
/****
 *
 * main(): Solicita um número inteiro positivo ao usuário e
 *         apresenta os números primos menores do que ou iguais
 *         ao valor introduzido usando o Crivo de Eratóstenes
 *
 * Parâmetros: Nenhum
 *
 * Retorno: 0, se não ocorrer nenhum erro; 1, em caso contrário.
 *
 ****/

int main(void)
{
   int numero;

      /* Apresenta o programa */
   printf( "\n\t>>> Este programa encontra os numeros primos que "
           "se\n\t>>> encontram entre 2 e o valor inteiro "
           "positivo que\n\t>>> voce introduzir.\n" );

      /* Lê o número */
   printf("\n\t>>> Digite o numero: ");
   numero = LeInteiro();

      /* Verifica se valor introduzido é válido e, se for */
      /* o caso, apresenta os números primos peneirados   */
   if (numero > 1) {
      if (Eratostenes(numero)) {
         printf("\n\t>>> Impossivel encontrar primos\n");
         return 1; /* Peneira de Eratóstenes está rasgada */
      }
   } else {
      printf("\n\t>>> O numero deve ser maior do que 1\n");
      return 1; /* Usuário não sabe o que é número primo */
   }

   printf( "\n\t>>> Obrigado por usar este programa e visite"
           "\n\t>>> o Nordeste para conhecer uma urupema.\n");

   return 0;
}
```

Análise: Com o conhecimento que você adquiriu até aqui, seria ultrajante apresentar uma análise dessa função **main**()...

Complemento do programa:

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

   /* Quantidade de números primos exibidos por linha */
#define PRIMOS_POR_LINHA  8
```

Exemplo de execução do programa:

```
        >>> Este programa encontra os numeros primos que se
        >>> encontram entre 2 e o valor inteiro positivo que
        >>> voce introduzir.

        >>> Digite o numero: 200

          *** Primos entre 2 e 200 ***

  2     3     5     7    11    13    17    19
 23    29    31    37    41    43    47    53
 59    61    67    71    73    79    83    89
 97   101   103   107   109   113   127   131
137   139   149   151   157   163   167   173
179   181   191   193   197   199

        >>> Obrigado por usar este programa e visite
        >>> o Nordeste para conhecer uma urupema.
```

12.7 Exercícios de Revisão

Introdução (Seção 12.1)

1. Cite cinco exemplos de programação apresentados em capítulos anteriores que seriam melhor resolvidos com o uso de alocação dinâmica de memória.
2. Cite três situações em programação nas quais alocação dinâmica de memória se faz necessária (além daquelas apontadas no exercício anterior).
3. Defina (a) variável estática e (b) variável dinâmica.
4. (a) O que é alocação estática de memória? (b) O que é alocação dinâmica de memória?
5. Quais dentre as variáveis no fragmento de programa a seguir são: (a) estáticas e (b) dinâmicas?

```
int           x;
static double y;
int*          p = malloc(sizeof(int));
```

6. (a) Variáveis definidas com **static** são consideradas variáveis estáticas? (b) E quanto a variáveis definidas sem **static**?

Funções de Alocação Dinâmica de Memória (Seção 12.2)

7. Descreva o funcionamento de cada uma das funções a seguir:
   (a) **malloc()**
   (b) **calloc()**
   (c) **free()**
8. (a) Quando o valor retornado por **realloc()** é **NULL**, o ponteiro passado como primeiro parâmetro continua válido? (b) Quando o valor retornado por **realloc()** é diferente de **NULL**, o ponteiro passado como primeiro parâmetro continua válido?
9. Como funciona a função **realloc()** quando o primeiro parâmetro é **NULL**?
10. Como funciona a função **realloc()** quando o valor do segundo parâmetro é zero?
11. Por que não é aconselhável atribuir o valor retornado por **realloc()** ao mesmo ponteiro passado como primeiro parâmetro numa chamada dessa função?
12. (a) O que é uma variável anônima? (b) Por que variáveis anônimas aparecem apenas no contexto de alocação dinâmica de memória?

13. Variáveis estáticas são liberadas automaticamente (v. **Seção 5.9**). Então, por que variáveis dinâmicas precisam ser liberadas explicitamente?
14. Descreva dois erros comuns de liberação de blocos alocados dinamicamente e como o programador pode precaver-se contra eles.
15. Suponha que `p` seja um ponteiro que correntemente esteja apontando para um bloco alocado dinamicamente e `n` seja um valor inteiro positivo. O que há de errado com a seguinte chamada de **realloc()**?

```
p = realloc(p, n*sizeof(*p));
```

16. A função `CriaArray()` a seguir foi implementada com a intenção de alocar dinamicamente um array de elementos do tipo **int**. Essa função chama `LeInteiro()` da biblioteca LeituraFacil para ler os valores dos elementos do array. O que há de (muito) errado com a função `CriaArray()`?

```
int CriaArray(int *array, int tamanho)
{
   int i;

   array = malloc(sizeof(int)*tamanho);

   if(!array) {
      return 1; /* Erro de alocação */
   }

   printf("Digite %d inteiros:\n", tamanho);

   for (i = 0; i < tamanho; ++i) {
      array[i] = LeInteiro();
   }

      /* Tudo ocorreu bem */
   return 0;
}
```

Ponteiros Genéricos e o Tipo void * (Seção 12.3)

17. O que é um ponteiro genérico?
18. Como ponteiros genéricos são definidos?
19. Em quais situações o tipo **void** * é normalmente utilizado?
20. Apresente dois exemplos de uso do tipo **void** *.
21. (a) Por que o programa a seguir não consegue ser compilado? (b) Como corrigir esse programa para que ele possa ser compilado?

```
#include <stdlib.h>

int main(void)
{
   void *p = malloc(sizeof(int));

   *p = 5;

   return 0;
}
```

Espaço de Execução de um Programa (Seção 12.4)

22. Como tipicamente é dividido o espaço reservado para a execução de um programa?
23. Qual é a importância de *heap* em alocação dinâmica de memória?

24. (a) O que significa fragmentação de *heap*? (b) Qual pode ser a consequência danosa decorrente da ocorrência de fragmentação de *heap*?

25. (a) O que é um zumbi de heap? (b) O que é um zumbi de pilha?

26. (a) O que é escoamento de memória? (b) Quais são os sintomas aparentes de um programa com escoamento de memória?

Testando Alocação Dinâmica (Seção 12.5)

27. (a) Como deve ser testado um endereço retornado por uma função de alocação dinâmica de memória? (b) Por que é sempre recomendado testar o endereço retornado por uma função de alocação dinâmica de memória?

28. Qual é o problema com o programa a seguir?

```
#include <stdlib.h>
#include <stdio.h>

#define TAMANHO 10

int main(void)
{
   int *ar, i;

   ar = malloc(TAMANHO*sizeof(int));

   for (i = 0; i < TAMANHO; i++)
      *(ar + i) = i * i;

   for (i = 0; i < TAMANHO; i++)
      printf("%d\n", *ar++);

   free(ar);

   return 0;
}
```

12.8 Exercícios de Programação

12.8.1 Fácil

EP12.1 Muitas extensões da biblioteca padrão de C oferecem uma função, comumente denominada `strdup()`, que cria, utilizando alocação dinâmica de memória, uma cópia de um string recebido como parâmetro. Implemente essa função, cujo protótipo é dado por:

```
char *strdup(const char*);
```

[**Sugestão**: Use **malloc()**, **strlen()** e **strcpy()**.]

EP12.2 Acrescente um pouco mais de dificuldade ao jogo de palavras implementado pelo programa apresentado na **Seção 12.6.2** requerendo que a palavra sorteada tenha pelo menos certo tamanho estabelecido por uma constante simbólica denominada `TAM_MIN_PALAVRA`. [**Sugestão**: Substitua o trecho da função **main()** que realiza o sorteio por uma chamada de uma função, denominada `SorteiaPalavra()`, que sorteia cada palavra usada no jogo. Essa função deve receber como parâmetros o número de palavras no arquivo e o tamanho mínimo que a palavra sorteada deve ter. Essa função deve retornar o endereço do array alocado dinamicamente por `LeLinhaIlimitada()` e que contém a palavra sorteada apenas quando essa palavra tiver o tamanho mínimo especificado.]

EP12.3 Considerando novamente o jogo de palavras apresentado na **Seção 12.6.2**, seja mais simpático com o usuário, criando um programa que permita que ele decida quais são os tamanhos mínimo e máximo da palavra sorteada. [**Sugestões**: (1) Modifique a função `SorteiaPalavra()` sugerida no problema

EP12.2, de modo que ela receba mais um parâmetro que represente o tamanho máximo da palavra a ser sorteada. Então, essa função deve retornar apenas quando o tamanho da palavra estiver entre os limites especificados por seus parâmetros. (2) Acrescente um trecho de programa à função **main()** responsável pela leitura dos limites mínimo e máximo de tamanho da palavra usada no jogo.]

EP12.4 Mais uma vez, considerando o jogo de palavras da Seção 12.6.2, criando um programa que permita que o usuário jogue quantas partidas ele desejar para que ele possa treinar para participar de um programa de TV. [**Sugestões**: (1) Use as sugestões apresentadas para os problemas EP12.2 e EP12.3. (2) Coloque o trecho do programa responsável pela interação com o usuário no corpo de um laço de repetição que encerrará apenas quando o usuário assim decidir. (3) Use a função `LeOpcaoSimNao()`, definida na Seção 5.8.3, para verificar se o usuário deseja continuar jogando até ficar exaurido.]

EP12.5 A função `CriaArquivoBin()`, apresentada na Seção 11.15.10, possui uma limitação decorrente da estimativa que ela faz com respeito ao maior tamanho de linha no arquivo de texto processado. Reescreva essa função, de forma a corrigir essa limitação. [**Sugestão**: Use a função `LeLinhaIlimitada()`, apresentada na Seção 12.6.1, em substituição a **fgets()**.]

EP12.6 (a) Escreva uma função que cria um array de elementos do tipo **int** limitado apenas pela quantidade de memória alocada para o programa, mas que, ao mesmo tempo, não desperdice memória. Os valores dos elementos do array devem ser lidos via teclado. (b) Escreva um programa que teste a função solicitada no item (a). [**Sugestão**: Utilize como modelo a implementação da função `LeLinhaIlimitada()` apresentada na Seção 12.6.1.]

EP12.7 Escreva um programa que exibe na tela seu próprio código-fonte (se o arquivo-fonte for encontrado, obviamente), levando em consideração o fato de o nome principal do programa ser o mesmo nome principal do programa executável. **NB**: Não há como determinar com certeza o nome do arquivo de texto que deu origem a um determinado programa executável. Portanto o melhor que se pode fazer é tentar adivinhá-lo usando intuição. [**Sugestões**: (1) A função **main()** desse programa deve possuir parâmetros, pois, assim, ela poderá saber o nome do programa executável. (2) Aloque dinamicamente um array para armazenar o nome do arquivo-fonte, cujo tamanho deve ser estimado com base no tamanho do nome do programa executável, que é recebido como argumento pelo programa. (3) Verifique se o nome do programa executável inclui o caractere `'/'` (família Unix) ou `'\'` (família Windows). Se for o caso, copie o nome do arquivo após o último caractere `'/'` (ou `'\'`) para o array. Caso contrário, copie todo o nome do programa para o array. (4) Se o arquivo executável tiver extensão, substitua-a por `'c'`. Caso contrário, acrescente os caracteres `'.'` e `'c'` ao seu nome. (5) Tente abrir o arquivo cujo nome foi armazenado no array para leitura em modo texto. (6) Se o arquivo não foi aberto, informe que o programa não conseguiu encontrar o arquivo-fonte e encerre. Caso contrário, apresente seu conteúdo na tela utilizando os conhecimentos adquiridos no Capítulo 11.]

EP12.8 Escreva uma função que recebe dois strings como entrada e retorna um ponteiro para um array alocado dinamicamente contendo o resultado da concatenação dos dois strings.

EP12.9 Escreva um programa que lê um arquivo de texto e escreve na tela apenas aquelas linhas que contêm um string indicado pelo usuário. [**Sugestões**: (1) Use `LeLinhaIlimitada()` (. Seção 12.6.1) para ler o string introduzido pelo usuário e as linhas do arquivo. (2) Use **strstr()** para checar quando o string citado está presente em cada linha lida.]

EP12.10 Reescreva o programa apresentado como exemplo na Seção 8.11.4 de modo que o array que armazena a maior sequência de Fibonacci seja alocado dinamicamente e não ocorra nem desperdício nem carência de memória.

EP12.11 (a) Escreva uma função que concatena um string um número especificado de vezes. [**Sugestões**: (1) Aloque dinamicamente um array com espaço suficiente para conter o resultado da concatenação.

Não esqueça do espaço adicional para o caractere `'\0'`. (2) Copie o string para esse array usando **strcpy()**. (3) Acrescente, sucessivamente, `n-1` cópias do string ao mesmo array usando **strcat()**.] (b) Escreva um programa para testar a função especificada no item (a). [**Sugestão**: Use a função `LeLinhaIlimitada()`, apresentada na **Seção 12.6.1**, para ler o string introduzido pelo usuário.]

12.8.2 Moderado

EP12.12 (a) Escreva uma função que remove todas as ocorrências de um string de um stream de texto e armazena o resultado em outro stream de texto. Essa função deve retornar o número de substituições efetuadas. [**Sugestões**: (1) Utilize a função `LeLinhaIlimitada()`, apresentada na **Seção 12.6.1**, para ler cada linha do arquivo de entrada. (2) Use **strstr()** para localizar as ocorrências e **strlen()** para determinar o tamanho do string a ser removido em cada linha lida. (3) Escreva no arquivo de saída, usando **fputc()**, os caracteres de cada linha, exceto aqueles que fazem parte do string a ser removido.] (b) Escreva um programa que recebe como argumentos de linha de comando dois nomes de arquivos de texto e um string que deve ser removido do primeiro arquivo, de modo que o resultado seja escrito no segundo arquivo. O programa deve ainda informar o número de substituições efetuadas, como mostra o seguinte exemplo de execução:

```
C:\Programas> RemoveString Tudor.txt TudorBK.txt Ana
        >>> Foram efetuadas 2 remocoes
```

EP12.13 (a) Escreva uma função que substitui todas as ocorrências de um string em um stream de texto por outro string e armazena o resultado em outro stream de texto. Essa função deve retornar o número de substituições efetuadas. [**Sugestões**: (1) Utilize a função `LeLinhaIlimitada()`, apresentada na **Seção 12.6.1**, para ler cada linha do arquivo de entrada. (2) Use **strstr()** para localizar as ocorrências e **strlen()** para determinar o tamanho do string a ser removido em cada linha lida. (3) Escreva no arquivo de saída, usando **fputc()**, os caracteres de cada linha, exceto aqueles que fazem parte do string a ser substituído. Em vez desses caracteres, escreva os caracteres do string substituto.] (b) Escreva um programa que substitui todas as ocorrências de um string de um arquivo de texto por outro string e escreve o resultado em outro arquivo de texto. O programa deverá receber como argumentos de linha de comando (nessa ordem): o nome do arquivo de entrada, o nome do arquivo que conterá as possíveis alterações, o string que será substituído e o string que irá substituí-lo.

Exemplo de execução do programa:

```
C:\Programas> SubsString Tudor.txt TudorBK.txt Ana Banana
        >>> Foram efetuadas 2 substituicoes de
        >>> "Ana" por "Banana" no arquivo "Tudor.txt"
```

12.9 Projetos de Programação

PP12.1 Implementação do Programa compila

Problema: Escreva um programa que auxilia compilação e ligação de programas via linha de comando usando opções de compilação/ligação armazenadas num arquivo e permite alterar essas opções. Esse programa deve manter um arquivo de configuração, denominado `compila.cfg`, que armazena opções de compilação usadas pelo usuário. O programa deve ainda funcionar em modo interativo e não interativo, como mostram os exemplos de execução a seguir:

Exemplo de execução do programa: Versão não interativa

```
Ulysses-iMac:~ ulysses$ ./compila Teste
   >>> Sistema: Mac OS X
   >>> Compilador utilizado: Clang
   >>> Arquivo de configuracao: Unix

   >>> O compilador sera' invocado com o seguinte comando:
      > clang -Wall -std=c99 Teste.c -lleitura -lm -o Teste

   >>> Deseja continuar (s/n)? s

   >>> Deseja que o programa continue solicitando confirmacao (s/n)? n
   --> Compilacao bem sucedida
```

Exemplo de execução do programa: Versão interativa

```
>>> Sistema: Windows
>>> Compilador utilizado: gcc
>>> Arquivo de configuracao: Windows

********** Opcoes *********
C. Compila Programa
A. Acrescenta Opcao de Compilacao/Ligacao
R. Remove Opcao de Compilacao/Ligacao
P. Apresenta Opcoes de Compilacao e Ligacao
Q. Arquivo de Configuracao(Reconstrucao)
J. Ajuda
E. Encerra Programa

>>> Escolha uma opcao: j

******** Topicos de Ajuda ********
A. Acrescenta Opcao de Compilacao/Ligacao
R. Remove Opcao de Compilacao/Ligacao
C. Compila Programa
E. Encerra a Ajuda

>>> Escolha uma opcao: a

> Digite a opcao iniciando sempre com '-'. Se desistir de
> acrescentar uma opcao, digite apenas [ENTER].

******** Topicos de Ajuda ********
A. Acrescenta Opcao de Compilacao/Ligacao
R. Remove Opcao de Compilacao/Ligacao
C. Compila Programa
E. Encerra a Ajuda
>>> Escolha uma opcao: r
> Escolha a opcao que deseja remover. Se desistir de remover,
> escolha a alternativa 'NENHUMA'.

******** Topicos de Ajuda ********
A. Acrescenta Opcao de Compilacao/Ligacao
R. Remove Opcao de Compilacao/Ligacao
C. Compila Programa
E. Encerra a Ajuda
>>> Escolha uma opcao: c
> Digite apenas o nome do programa-fonte sem extensao. Assume-se
> que a extensao do arquivo e' ".c". Se desistir de compilar
> um arquivo, digite apenas [ENTER].
```

```
******** Topicos de Ajuda ********
A. Acrescenta Opcao de Compilacao/Ligacao
R. Remove Opcao de Compilacao/Ligacao
C. Compila Programa
E. Encerra a Ajuda
>>> Escolha uma opcao: e

********** Opcoes *********
C. Compila Programa
A. Acrescenta Opcao de Compilacao/Ligacao
R. Remove Opcao de Compilacao/Ligacao
P. Apresenta Opcoes de Compilacao e Ligacao
Q. Arquivo de Configuracao(Reconstrucao)
J. Ajuda
E. Encerra Programa

>>> Escolha uma opcao: e
```

Sugestões:

[1] Provavelmente, as seguintes funções definidas neste livro o ajudarão na implementação deste projeto:

Função	Seção
`LeOpcaoSimNao()`	5.8.3
`FechaArquivo()`	11.5
`NumeroDeLinhas()`	11.15.5
`CopiaArquivo()`	11.15.6
`LeLinhaIlimitada()`	12.6.1

[2] As funções que aparecem com suas descrições apresentadas na tabela a seguir precisarão ser implementadas.

Função	Descrição
`ClassificaArquivo()`	Classifica um arquivo de texto de acordo com sua família de quebra de linha
`WindowsParaUnix()`	Converte um arquivo de texto de formato Windows para formato Unix
`UnixParaWindows()`	Converte um arquivo de texto de formato Unix para formato Windows
`ChecaSistema()`	Tenta determina o sistema no qual o programa é executado
`ChecaConfiguracao()`	Determina o tipo de sistema no qual o programa está sendo executado
`Compilador()`	Retorna o nome do compilador utilizado
`CriaArquivoConfig()`	Cria um arquivo de configuração
`CriaComando()`	Cria um comando de compilação
`CompilaPrograma()`	Compila um programa

Função	Descrição
`ApresentaOpcoes()`	Apresenta as opções de compilação ou ligação contidas no arquivo de configuração
`AcrescentaOpcao()`	Acrescenta uma opção de compilação ou ligação ao arquivo de configuração
`RemoveTokenDeString()`	Remove de um string o token cujo índice é especificado
`RemoveOpcao()`	Remove uma opção de compilação ou ligação do arquivo de configuração
`LeOpcaoDeConfirmacao()`	Lê no arquivo de configuração a opção utilizada para confirmação de um comando
`EscreveOpcaoDeConfirmacao()`	Escreve no arquivo de configuração a opção utilizada para confirmação de um comando
`Ajuda()`	Oferece assistência ao usuário
`SaltaLinhas()`	Salta um número especificado de linhas de um arquivo de texto