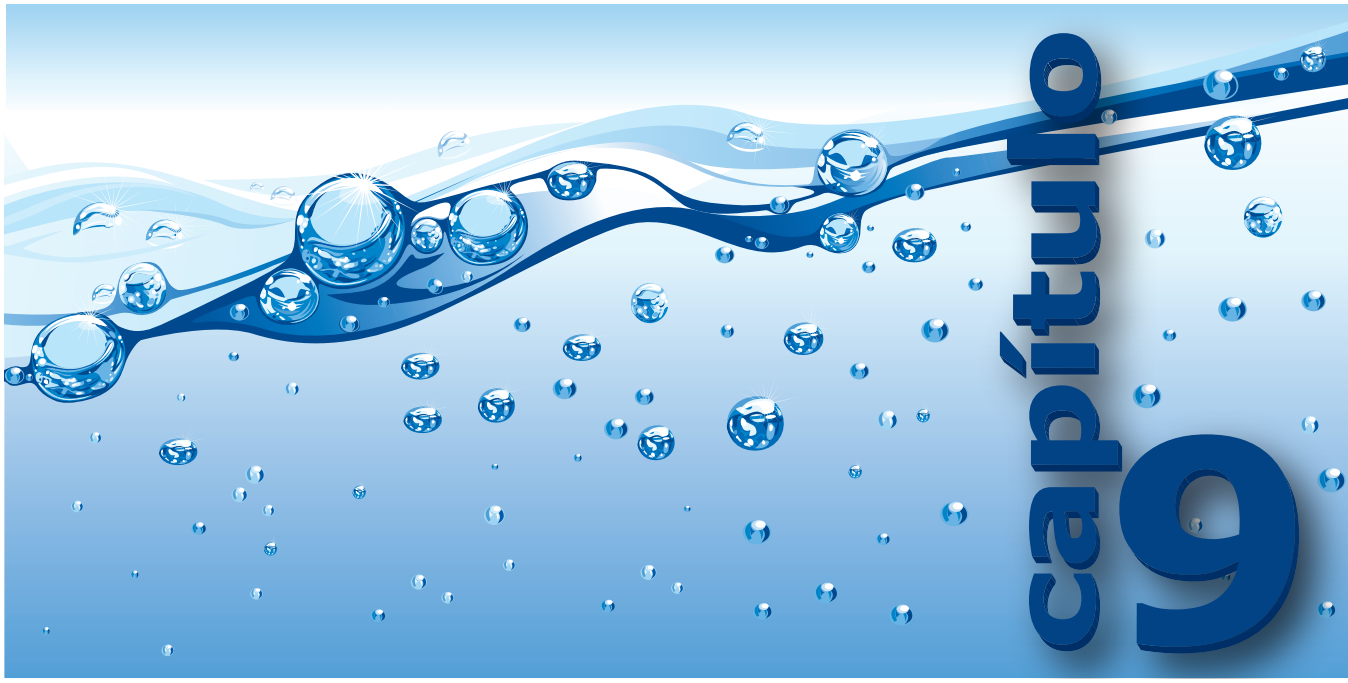

Alocação Dinâmica de Memória

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
- 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
- Explicar como a macro **ASSEGURA** pode ser usada para garantir que uma chamada de função de alocação dinâmica de memória foi bem-sucedida
- Implementar o TAD lista indexada usando array dinâmico
- Implementar os TADs pilha e fila usando array dinâmico
- Mostrar o papel desempenhado por **realloc()** no processamento de listas dinâmicas
- Explicar a necessidade de função de destruição para estrutura de dados dinâmica
- Mostrar que os custos temporal e espacial da função **realloc()** são $\theta(n)$ no pior caso
- Implementar uma função capaz de ler linhas de qualquer tamanho num stream de texto

objetivos

Neste capítulo, os conceitos de alocação estática e dinâmica de memória serão apresentados. Do ponto de vista prático, este capítulo mostra como alocar e liberar memória dinamicamente por meio de chamadas de funções da biblioteca padrão de C e como implementar estruturas de dados por meio dessa técnica.

9.1 Motivação e Justificativas

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 2.3**). Em ambas as formas de alocação de memória se assume 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 explorado 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.

Todas as implementações de estruturas de dados apresentadas em capítulos anteriores sofrem com essa falta de capacidade de alocar memória precisamente. Em todas elas ocorre superdimensionamento (se ocorresse subdimensionamento, como elas iriam funcionar?). Por exemplo, nas implementações de listas, pilhas e filas, utilizou-se uma constante simbólica, denominada `MAX_ELEMENTOS`, que é uma estimativa do número máximo de elementos que serão armazenados nessas estruturas de dados. Nesses casos, os programas que são clientes dessas estruturas de dados ora podem desperdiçar memória, em virtude de superdimensionamento, ora podem deixar de funcionar adequadamente, por causa de subdimensionamento. Assim essas estruturas de dados precisam ser dotadas de capacidade para alocar memória de acordo com a demanda apresentada durante seus usos.

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.

9.2 Funções de Alocação e Liberação Dinâmica de Memória

Alocação e liberação dinâmica de memória em C é realizada por meio de ponteiros e quatro funções de biblioteca resumidas na **Tabela 9–1**. 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 9–1: Funções de Alocação e Liberação Dinâmica de Memória

Essas funções serão exploradas em profundidade adiante, mas, antes de prosseguir, é oportuno definir o conceito de **bloco de memória**, que é um conjunto de bytes contíguos em memória. O tipo **size_t** (v. **Seção 1.16.3**) é tipicamente utilizado para calcular tamanhos de blocos de memória (v. adiante). Doravante, neste capítulo, bloco será utilizado como sinônimo de bloco de memória.

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.

9.2.1 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 7.6.2**, 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 9.3**. 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 9.4**).

9.2.2 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.

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.

9.2.3 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.

9.2.4 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 9–1** e a **Figura 9–2** 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 9–1**), ao passo que, 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 9–2**).

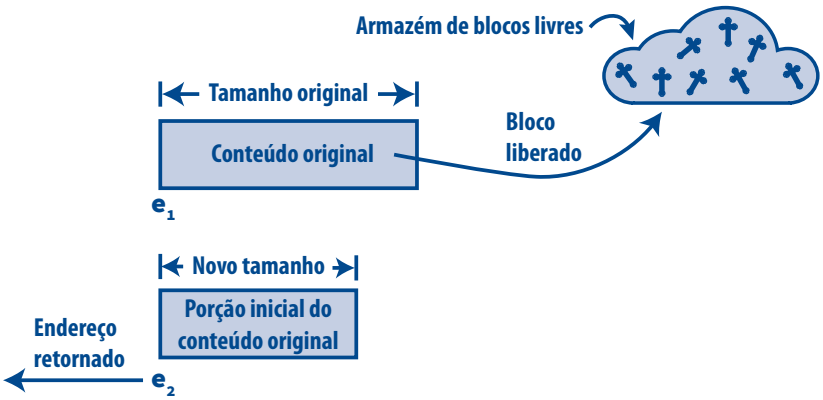

FIGURA 9–1: FUNÇÃO REALLOC(): NOVO BLOCO É MENOR DO QUE O BLOCO ORIGINAL

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 num novo endereço nem seu tamanho foi alterado. Em outras palavras, a chamada de **realloc()** foi absolutamente ineficaz. Deve-se ressaltar que, nessa situação, o primeiro 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.

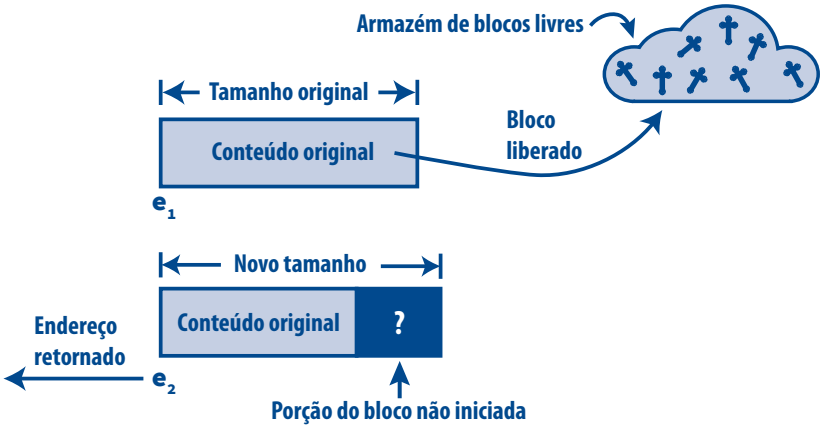

FIGURA 9–2: FUNÇÃO REALLOC(): NOVO BLOCO É MAIOR DO QUE O BLOCO ORIGINAL

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:

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.

As Seções **9.6.1**, **9.6.2** e **9.9** apresentarão exemplos de uso prático de **realloc()**.

9.3 Ponteiros Genéricos e o Tipo void *

Recorde-se que dois ponteiros são compatíveis apenas quando eles são exatamente do mesmo tipo (v. **Seção 1.18**). 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 quaisquer tipos de ponteiros. Por exemplo, o parâmetro formal da função **free**() (v. **Seção 9.2**) 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 ExibeValor(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;

   ExibeValor(&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 permitida nenhuma operação de aritmética de ponteiros (v. **Seção 3.2**) sobre um ponteiro genérico que não tenha sido convertido explicitamente para um tipo de ponteiro conhecido.

9.4 A Partição de Memória Heap

O espaço reservado em memória para a execução de um programa é dividido em partições, conforme foi discutido na **Seção 4.3**. Aqui, a ênfase será no heap, que é a partição na qual ocorre alocação dinâmica de memória, como mostra a **Figura 9–3**.

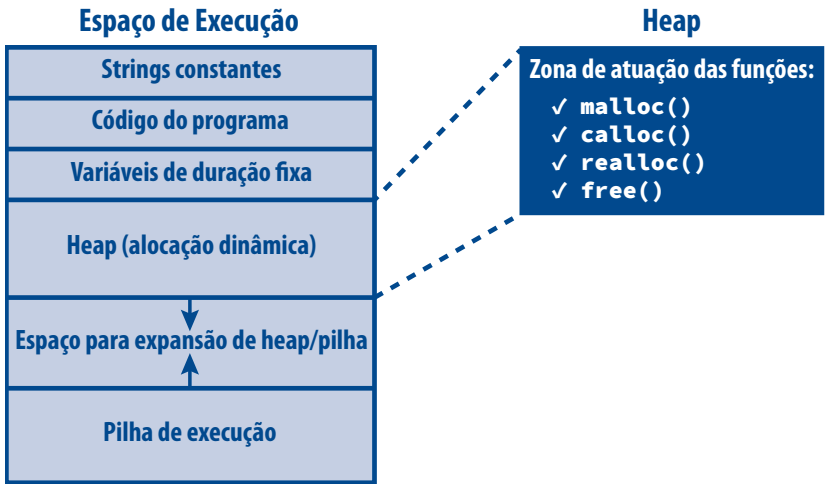

Figura 9–3: Espaço de Execução de Programa 2

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 de um 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 foi compilado com GCC e executado em Windows XP, o programa apresentou 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.

9.5 Testando Alocação Dinâmica de Memória

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, 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.

A macro `ASSEGURA`, apresentada na **Seção 7.4**, também pode ser utilizada para checar alocação dinâmica de memória, como mostrado a seguir:

```
ASSEGURA( ptrAluno = malloc(sizeof(tAluno)), "Nao foi possivel alocar um bloco" );
```

9.6 Listas Indexadas Dinâmicas

Lista é um conceito abstrato que pode ser implementado de diversas maneiras. O modo mais simples de implementação de listas é por meio de arrays estáticos, conforme foi discutido no **Capítulo 7**. No presente capítulo, será mostrado como implementar listas sem ordenação e listas ordenadas como tipos abstratos de dados (TADs — v. **Capítulo 5**) usando arrays dinâmicos.

Existem duas abordagens básicas de implementação de listas usando arrays dinâmicos:

[1] Manter o array utilizado para armazenar os elementos de uma lista do tamanho exatamente igual ao número de elementos presentes na lista em qualquer instante. Aparentemente, essa é a melhor abordagem, pois não ocorre nenhum desperdício de memória. Ocorre, porém, que esse não é o caso, pois a cada inserção e remoção de elemento será requerida uma chamada de **realloc()** e cada chamada dessa função tem, no pior caso, custo $\theta(n)$, em que n é o número de bytes do bloco sendo redimensionado. Mesmo assim, com objetivo meramente didático, essa abordagem será demonstrada na **Seção 9.6.1**.

[2] Manter sempre espaço sobressalente no array que armazena os elementos da lista, de modo que chamadas de **realloc()** sejam necessárias apenas quando esse espaço estiver exaurido. Agora, quando ocorre uma remoção, essa função é chamada apenas quando o tamanho do array atingir um determinado patamar que seja interpretado como desperdício de memória. Essa abordagem, relativamente falando, é o oposto da primeira. Quer dizer, ela parece causar desperdício de memória, o que de fato ocorre, mas durante a execução de uma chamada de **realloc()** o desperdício pode ser bem maior (v. **Seção 10.1**). Essa abordagem será implementada na **Seção 9.6.2**.

Nos exemplos que serão apresentados a seguir, o tipo de cada elemento de uma lista é `tAluno`, que foi apresentado na **Seção 7.6.2**.

9.6.1 Lista sem Ordenação

Esta seção discute a primeira abordagem de implementação de listas como TADs que usam arrays dinâmicos para armazenamento dos elementos das listas. Nessa abordagem, o tamanho de uma lista coincide com o tamanho do array que armazena seus elementos. É importante lembrar que essa não é a melhor forma de uso de arrays dinâmicos, pois ela requer um número excessivo de chamadas de **realloc()**. Portanto o objetivo aqui é apenas didático.

Definições de Tipos e Constantes

O cabeçalho do TAD em questão contém as seguintes definições de tipos:

```
typedef tAluno tElemento;

typedef struct rotListaIdxD *tListaIdxD;
```

Note que o tipo `tListaIdxD` é um tipo de ponteiro opaco (v. **Seção 5.4**) que aponta para uma estrutura incompleta.

Além das definições acima, o arquivo de cabeçalho contém alusões às funções globais definidas no arquivo de programa correspondente. A **Seção 2.1** mostra como escrever alusões de funções.

O arquivo de programa (implementação) contém o complemento do tipo `tListaIdxD` e as definições de função, nessa ordem. Esse complemento de tipo é apresentado a seguir:

```
struct rotListaIdxD {
   tElemento *elementos; /* Ponteiro para o array que contém os elementos */
   int        nElementos; /* Número de elementos */
};
```

As funções definidas no arquivo de implementação do TAD que diferem substancialmente daquelas funções que implementam operações equivalentes apresentadas no **Capítulo 7** serão vistas a seguir. A implementação completa desse TAD e de um programa-cliente que o usa encontra-se no site dedicado ao livro.

Criação

A função `CriaListaIdxD()` aloca espaço para a estrutura que armazena uma lista apontada por um ponteiro do tipo `tListaIdxD`. Ela ainda inicia o número de elementos na lista com zero e o ponteiro para o array que

armazenará os elementos com **NULL**. Essa função retorna o endereço da referida estrutura e utiliza a macro `ASSEGURA` discutida na **Seção 7.4**.

```
tListaIdxD CriaListaIdxD(void)
{
   tListaIdxD umaLista;

      /* Aloca espaço para a estrutura que armazena a lista */
   umaLista = malloc(sizeof(struct rotListaIdxD));

      /* Garante que a alocação realmente ocorreu */
   ASSEGURA(umaLista, "Nao foi possivel alocar lista");

      /* Inicialmente a lista está vazia */
   umaLista->nElementos = 0;
   umaLista->elementos = NULL;

   return umaLista;
}
```

Destruição

Conforme foi discutido na **Seção 5.4**, um TAD requer uma função que libere o espaço ocupado pela estrutura que representa a lista. Além disso, a função `DestroiListaIdxD()` apresentada a seguir libera o espaço ocupado pelo array que contém os elementos da lista.

```
void DestroiListaIdxD(tListaIdxD lista)
{
      /* Se a lista não estiver vazia, libera o espaço */
      /* ocupado pelo array que contém os elementos    */
   if (lista->nElementos)
      free(lista->elementos);

      /* Libera a estrutura que representa a lista */
   free(lista);
}
```

Inserção

Na abordagem em discussão, o tamanho da lista coincide com o tamanho do array, de modo que nunca há espaço sobressalente no array. Logo redimensionamento sempre se faz necessário quando ocorre inserção ou remoção de elementos.

A função `InsereListaIdxD()`, que será vista a seguir, sempre tenta redimensionar o array que armazena os elementos da lista para que ele possa conter mais um elemento.

```
void InsereListaIdxD(tListaIdxD lista, tElemento item, int indice)
{
   tElemento *novoArray; /* Ponteiro para o array redimensionado */
   int        nBytesArray; /* Número de bytes do array após inserção */

      /* Verifica se o índice é válido */
   ASSEGURA( indice >= 0 && indice <= lista->nElementos,
             "\nPosicao de insercao inexistente\n");

      /* Calcula o novo tamanho do array em bytes */
   nBytesArray = (lista->nElementos + 1)*sizeof(tElemento);

      /* Tenta redimensionar o array */
   novoArray = realloc(lista->elementos, nBytesArray);
```

```
      /* Garante que houve realocação de memória */
   ASSEGURA(novoArray, "Nao houve redimensionamento");

      /* O ponteiro que apontava para o início do array pode   */
      /* não ser mais válido.  Portanto é preciso atualizá-lo. */
   lista->elementos = novoArray;

      /* Abre espaço para o novo elemento */
   for (int i = lista->nElementos - 1; i >= indice; --i)
      lista->elementos[i + 1] = lista->elementos[i];

   lista->elementos[indice] = item; /* Insere o novo elemento */

   lista->nElementos++; /* O tamanho da lista aumentou */
}
```

Observe que, quando a chamada de **realloc**() na função acima é bem-sucedida, o ponteiro que antes apontava para o início do bloco que foi realocado pode agora estar apontando para um bloco que foi liberado, conforme foi discutido na **Seção 9.2**. Portanto esse ponteiro precisa ser atualizado com o valor do ponteiro que recebeu o endereço retornado por **realloc**(). Precisamente, no caso em questão, após a chamada de **realloc**():

```
novoArray = realloc(lista->elementos, nBytesArray);
```

é necessário atualizar o valor do ponteiro `lista->elementos` como:

```
lista->elementos = novoArray;
```

Remoção

A função `RemoveListaIdxD()` definida abaixo realiza uma operação de remoção de um elemento de uma lista de elementos do tipo `tAluno` implementada como TAD usando array dinâmico. Essa função retorna o elemento removido.

```
tElemento RemoveListaIdxD(tListaIdxD lista, int indice)
{
   tElemento  itemRemovido; /* Armazena o item removido */
   tElemento *novoArray; /* Ponteiro para o array redimensionado */
   int        nBytesArray; /* Número de bytes do array após inserção */

         /* Garante que o índice é válido */
   ASSEGURA(indice >= 0 && indice < lista->nElementos, "Posicao inexistente");

      /* Guarda o valor do elemento a ser removido */
   itemRemovido = lista->elementos[indice];

      /* Remover um elemento significa mover cada elemento uma posição */
      /* para trás a partir do sucessor do elemento que será  removido */
   for (int i = indice; i < lista->nElementos - 1; i++)
      lista->elementos[i] = lista->elementos[i + 1];

   --lista->nElementos; /* O tamanho da lista diminuiu */

      /* Nessa abordagem, o tamanho da lista coincide com o tamanho do array, de */
      /* modo que nunca há espaço desocupado no array e de modo que redimensio-  */
      /* namento sempre se faz necessário quando ocorre inserção ou remoção      */
      /*                                                                         */

      /* Calcula o novo tamanho do array em bytes */
   nBytesArray = lista->nElementos*sizeof(tElemento);

      /* Tenta redimensionar o array */
   novoArray = realloc(lista->elementos, nBytesArray);
```

```
      /* Garante que houve realocação de memória */
   ASSEGURA(novoArray, "Nao houve redimensionamento");

      /* O ponteiro que apontava para o início do array pode não */
      /* ser mais válido. Portanto é preciso atualizá-lo.        */
   lista->elementos = novoArray;

   return itemRemovido;
}
```

Acréscimo de Elemento

Quando uma lista implementada por meio de array não é ordenada, acrescentar um elemento ao final da lista é o modo mais fácil de implementar essa operação, como mostra a função `AcrescentaListaIdxD()` apresentada a seguir.

```
void AcrescentaListaIdxD(tListaIdxD lista, const tElemento *elemento)
{
   tElemento *novoArray; /* Ponteiro para o array redimensionado */
   int        nBytesArray; /* Número de bytes do  array após inserção */

      /* Calcula o novo tamanho do array em bytes */
   nBytesArray = (lista->nElementos + 1)*sizeof(tElemento);

      /* Tenta redimensionar o array */
   novoArray = realloc(lista->elementos, nBytesArray);

      /* Garante que houve realocação de memória */
   ASSEGURA(novoArray, "Nao foi possivel redimensionar o array");

      /* O ponteiro que apontava para o início do array pode   */
      /* não ser mais válido.  Portanto é preciso atualizá-lo. */
   lista->elementos = novoArray;

      /* Acrescenta o novo elemento ao final da lista */
   lista->elementos[lista->nElementos] = *elemento;

   ++lista->nElementos; /* O tamanho da lista aumentou */
}
```

9.6.2 Lista Ordenada

Esta seção discute a segunda abordagem de implementação de listas como TADs que usam arrays dinâmicos para armazenamento dos elementos da lista. Nessa abordagem, o array que armazena os elementos da lista conta, sempre que possível, com espaço sobressalente, de modo que chamadas de **realloc()** para redimensioná-lo são necessárias apenas quando esse espaço é exaurido. Além disso, nessa abordagem, quando ocorre uma remoção, essa função é chamada apenas quando o tamanho do array atinge um valor que seja interpretado como desperdício. Além disso, agora, as listas serão consideradas ordenadas pelo campo `nome` do tipo `tAluno` (v. **Seção 7.6.2**).

Definições de Tipos e Constantes

O arquivo de interface deste TAD difere significativamente daquele apresentado na **Seção 9.6.1** apenas com relação às alusões de funções. Precisamente, nessa interface não há alusões às funções `InsereListaIdxD()`, `AcrescentaListaIdxD()` e `AlteraElementoListaIdxD()`, pois essas funções não fazem sentido quando a lista é ordenada, conforme foi visto na **Seção 7.2**. Por outro lado, esse arquivo inclui alusão às funções `InsereEmOrdemIdxD2()` e `BuscaBinariaListaIdxD2()`. Outras diferenças dizem respeito ao uso do sufixo 2 em alguns identificadores para torná-los distintos de identificadores usados na **Seção 9.6.1**. Por exemplo, na presente seção, usa-se o tipo `tListaIdxD2`, em vez do tipo `tListaIdxD` usado na **Seção 9.6.1**.

Para implementar essa nova abordagem de uso de arrays dinâmicos, é necessária a seguinte definição de constante simbólica:

```
#define ACRESCIMO 10
```

Essa constante indica quantos elementos serão acrescentados ao array que armazena os elementos da lista sempre que sua capacidade de armazenamento estiver esgotada. Por simplicidade, essa mesma constante será usada para determinar quando ocorre desperdício de memória que seja considerado excessivo.

O arquivo de programa (implementação) contém o complemento do tipo `tListaIdxD2` e as definições de função, nessa ordem. Na implementação corrente, uma lista deve também incluir um campo que indique qual é o tamanho do array que armazena seus elementos, uma vez que, agora, esse tamanho não é sempre igual ao número de elementos da lista. Assim o complemento do tipo de lista é declarado como:

```
struct rotListaIdxD2 {
   tElemento *elementos; /* Ponteiro para o array que conterá os elementos da lista */
   int        nElementos;   /* Número de elementos da lista */
   int        tamanhoArray; /* Número de elementos do array */
};
```

O campo `tamanhoArray` de uma lista desse novo tipo armazena o tamanho do array utilizado pela lista. Ele é iniciado com zero e é atualizado sempre que o array é redimensionado.

As funções definidas no arquivo de implementação deste TAD que diferem daquelas funções correspondentes vistas na **Seção 9.6.1** serão apresentadas a seguir.

Criação

A função `CriaListaIdxD2()` aloca espaço para a estrutura que armazena uma lista apontada por um ponteiro do tipo `tListaIdxD2`. Essa função difere da função `CriaListaIdxD()` apresentada na **Seção 9.6.1** pelo fato de alocar um array mesmo quando não há nenhum elemento a ser incluído na lista.

```
tListaIdxD2 CriaListaIdxD2(void)
{
   tListaIdxD2 umaLista;

      /* Aloca espaço para a estrutura que armazena a lista */
   umaLista = malloc(sizeof(struct rotLista2));

      /* Garante que a alocação realmente ocorreu */
   ASSEGURA(umaLista, "Nao foi possivel alocar lista");

      /* Inicialmente a lista está vazia */
   umaLista->nElementos = 0;
   umaLista->elementos = NULL;

      /* Aloca previamente um array */
   umaLista->elementos = calloc(ACRESCIMO, sizeof(tElemento));

      /* Garante que o array foi realmente alocado */
   ASSEGURA( umaLista->elementos, "Nao foi possivel alocar "
             "array para conter elementos da lista" );

      /* Atualiza o campo que armazena o tamanho do array */
   umaLista->tamanhoArray = ACRESCIMO;

   return umaLista;
}
```

Inserção em Ordem

A função `InsereEmOrdemIdxD2()` insere um novo elemento numa lista do tipo em discussão de modo a preservar a ordenação da lista.

```
void InsereEmOrdemIdxD2(tListaIdxD2 lista, const tElemento *elemento)
{
   int        posicao; /* Posição de inserção do elemento */
   tElemento *novoArray; /* Ponteiro para o array redimensionado */

      /* Se o array estiver repleto, tenta redimensioná-lo */
   if (lista->nElementos >= lista->tamanhoArray) {
      lista->tamanhoArray += ACRESCIMO; /* Calcula o novo tamanho do array */

         /* Tenta redimensionar o array */
      novoArray = realloc(lista->elementos, lista->tamanhoArray*sizeof(tElemento));

         /* Checa se houve realocação de memória */
      ASSEGURA(novoArray, "Nao houve redimensionamento");

         /* O ponteiro que apontava para o início do array pode  */
         /* não ser mais válido. Portanto é preciso atualizá-lo. */
      lista->elementos = novoArray;
   }

      /* Se a lista estiver vazia, o elemento é */
      /* inserido na primeira posição do array  */
   if (EstaVaziaListaIdxD2(lista)) {
         /* O elemento será o primeiro da lista */
      lista->elementos[0] = *elemento;

      ++lista->nElementos; /* A lista cresceu */

      return;
   }

      /* Encontra a posição no array onde o elemento será inserido */
   for (posicao = 0; posicao < lista->nElementos; ++posicao)
      if ( strcmp(lista->elementos[posicao].nome, elemento->nome) > 0 )
         break; /* Posição de inserção encontrada */

      /* Abre espaço para o novo elemento */
   for (int i = lista->nElementos - 1; i >= posicao; --i) {
         /* Move cada elemento uma posição adiante a partir do  */
         /* elemento que ora se encontra na posição de inserção */
      lista->elementos[i + 1] = lista->elementos[i];
   }

   lista->elementos[posicao] = *elemento; /* Insere o novo elemento */
   ++lista->nElementos; /* A lista cresceu */
}
```

Diferentemente do que ocorre com a função `InsereListaIdxD()` discutida na **Seção 9.6.1**, nem sempre a função `InsereEmOrdemIdxD2()` redimensiona o array que armazena os elementos da lista. Além disso, quando o redimensionamento ocorre, o acréscimo no número de elementos do array é determinado pela constante simbólica `ACRESCIMO`.

Remoção

A função `RemoveListaIdxD2()` definida abaixo realiza uma operação de remoção de elemento de uma lista de elementos do tipo em discussão sem redimensionar o array que armazena os elementos da lista [como faz

a função `RemoveListaIdxD()` vista na seção anterior]. Essa ausência de redimensionamento pode fazer com que haja um certo desperdício de memória, mas, de todo modo, essa função é mais eficiente do que a função `RemoveListaIdxD()`.

```
tElemento RemoveListaIdxD2(tListaIdxD2 lista, int indice)
{
   tElemento itemRemovido;

         /* Verifica se o índice é válido */
   ASSEGURA( indice >= 0 && indice < lista->nElementos,
             "Posicao de remocao inexistente" );

   itemRemovido = lista->elementos[indice];

      /* Remover um elemento significa mover cada elemento uma posição */
      /* para trás a partir do sucessor do elemento que será  removido */
   for (int i = indice; i < lista->nElementos - 1; i++)
      lista->elementos[i] = lista->elementos[i + 1];

   --lista->nElementos; /* O tamanho da lista diminuiu */

   return itemRemovido;
}
```

9.7 Pilhas e Filas Implementadas com Arrays Dinâmicos

Esta seção mostra como os conceitos de pilha e fila estudados no **Capítulo 8** podem ser implementados como um tipo abstrato de dado usando arrays dinâmicos.

9.7.1 TAD Pilha 1

Nesta seção, o conceito de pilha será implementado como um TAD usando array dinâmico. Aqui, para redimensionamento de um array que armazena itens de uma pilha, será adotada a segunda abordagem discutida na **Seção 9.6**.

Arquivo de Cabeçalho

O arquivo de cabeçalho do TAD contém a definição do tipo de item a ser armazenado numa pilha desse tipo e uma definição incompleta do tipo, conforme foi visto na **Seção 5.4**. Para completar, o arquivo de cabeçalho contém as alusões das funções que representam as operações sobre pilhas.

```
/****************** Definições de Tipos ******************/

typedef  char tItemPilha;

typedef struct rotPilhaIdxD *tPilhaIdxD;

/****************** Alusões de Funções ******************/

extern void CriaPilhaIdxD(tPilhaIdxD *p);
extern void DestroiPilhaIdxD(tPilhaIdxD *pilha);
extern int EstaVaziaPilhaIdxD(const tPilhaIdxD *p);
extern tItemPilha ElementoTopoIdxD(const tPilhaIdxD *p);
extern void EmpilhaIdxD(tPilhaIdxD *p, const tItemPilha *item);
extern tItemPilha DesempilhaIdxD(tPilhaIdxD *p);
```

Note que as funções aludidas neste arquivo de cabeçalho apresentam exatamente os mesmos protótipos daquelas discutidas na **Seção 8.1**.

Arquivo de Implementação

O arquivo de implementação do TAD em questão não será apresentado na íntegra. Quer dizer, aqui serão discutidos apenas aqueles componentes que acrescentem algum conhecimento novo com relação ao que já foi visto com relação a pilhas, TADs e arrays dinâmicos.

Antes das definições das funções do referido arquivo de implementação, a constante simbólica `ACRESCIMO` deve ser definida. O complemento do tipo `tPilhaIdxD` segue as definições desses componentes e é apresentada abaixo.

```
struct rotPilhaIdxD {
   tItemPilha *itens; /* Ponteiro para o array que conterá os itens */
   int         topo;  /* Indicará o topo de uma pilha */
   int         tamanhoArray; /* Número de elementos do array */
};
```

A seguir, serão apresentadas as definições das funções `CriaPilhaIdxD()`, `DestroiPilhaIdxD()` e `EmpilhaIdxD()`, que são substancialmente diferentes das funções semelhantes apresentadas na **Seção 8.1**. As funções `EstaVaziaPilhaIdxD()` e `ElementoTopoIdxD()` são implementadas aqui exatamente do mesmo modo visto naquela seção. Se a abordagem de redimensionamento de array dinâmico fosse a primeira descrita na **Seção 9.6**, a função `DesempilhaIdx()` na **Seção 8.1** precisaria ser modificada, mas esse não é o caso aqui.

```
void CriaPilhaIdxD(tPilhaIdxD *umaPilha)
{
      /* Aloca espaço para a estrutura que armazena a pilha */
   *umaPilha = malloc(sizeof(struct rotPilhaIdxD));

      /* Garante que a alocação realmente ocorreu */
   ASSEGURA(*umaPilha, "Nao foi possivel alocar pilha");

   (*umaPilha)->topo = -1; /* Inicia o topo da pilha */

      /* Aloca um array inicial */
   (*umaPilha)->itens = calloc(ACRESCIMO, sizeof(tItemPilha));

      /* Garante que o array foi realmente alocado */
   ASSEGURA( (*umaPilha)->itens, "Nao foi possivel alocar "
             "array para conter elementos da pilha" );

      /* Atualiza a variável que armazena o tamanho do array */
   (*umaPilha)->tamanhoArray = ACRESCIMO;
}
```

Note que a função `CriaPilhaIdxD()` efetua duas alocações de memória:

[1] A chamada de **malloc()** é responsável pela alocação da estrutura que representa a pilha. Uma vez alocada, essa estrutura permanece com seu tamanho fixo durante a utilização da pilha.

[2] A chamada de **calloc()** aloca o array que armazenará os itens da pilha. Esse array poderá ter seu tamanho alterado de acordo com a utilização da pilha.

```
void DestroiPilhaIdxD(tPilhaIdxD *pilha)
{
      /* Se a pilha não estiver vazia, libera o espaço */
      /* ocupado pelo array que contém os elementos    */
   if ((*pilha)->topo)
      free((*pilha)->itens);

   free((*pilha)); /* Libera a estrutura que representa a pilha */
}
```

A função `DestroiPilhaIdxD()` efetua duas chamadas de **free()**, sendo uma para cada espaço que foi alocado dinamicamente.

```
void EmpilhaIdxD(tPilhaIdxD *p, const tItemPilha *item)
{
   tItemPilha *novoArray; /* Ponteiro para o array redimensionado */

      /* Se o array estiver repleto, tenta redimensioná-lo */
   if ((*p)->topo >= (*p)->tamanhoArray) {
         /* Calcula o novo tamanho do array */
      (*p)->tamanhoArray += ACRESCIMO;

         /* Tenta redimensionar o array */
      novoArray = realloc((*p)->itens, (*p)->tamanhoArray*sizeof(tItemPilha));

         /* Checa se houve realocação de memória */
      ASSEGURA(novoArray, "Nao houve redimensionamento");

         /* O ponteiro que apontava para o início do array pode   */
         /* não ser mais válido.  Portanto é preciso atualizá-lo. */
      (*p)->itens = novoArray;
   }

   (*p)->itens[++(*p)->topo] = *item;  /* Efetua o empilhamento */
}
```

Observe que, no arquivo de implementação acima, não existe a função privada para testar se a pilha está cheia. De fato, ele é desnecessário aqui pois, com alocação dinâmica de memória, a pilha pode crescer arbitrariamente (é claro, dentro das limitações impostas pela memória alocada para a execução do programa).

9.7.2 TAD Fila 1

Esta seção mostra como implementar o conceito de fila como um TAD que usa array dinâmico para armazenamento de seus itens. Novamente, será adotada a segunda abordagem para redimensionamento de arrays dinâmicos discutida na **Seção 9.6**.

Arquivo de Cabeçalho

O arquivo de cabeçalho do TAD contém a definição do tipo de item a ser armazenado em filas desse tipo e uma definição incompleta do tipo. As alusões das funções que representam as operações sobre filas completam o arquivo de cabeçalho. Essas alusões apresentam exatamente os mesmos protótipos das funções discutidas na **Seção 8.2**.

```
/****************** Definições de Tipos ******************/

typedef char tItemFila;

typedef struct rotFilaIdxD *tFilaIdxD;

/******************* Alusões de Funções ******************/

extern void CriaFilaIdxD(tFilaIdxD *f);
extern void DestroiFilaIdxD(tFilaIdxD *fila);
extern int EstaVaziaFilaIdxD(const tFilaIdxD *f);
extern tItemFila ElementoFrenteIdxD(const tFilaIdxD *f);
extern void EnfileiraIdxD(tFilaIdxD *f, const tItemFila *item);
extern tItemFila DesenfileiraIdxD(tFilaIdxD *f);

#endif
```

Arquivo de Implementação

Novamente, a constante simbólica `ACRESCIMO` discutida na **Seção 9.6.2** é definida antes das definições de funções. O complemento do tipo `tFilaIdxD` é apresentado abaixo.

```
struct rotFilaIdxD {
   tItemFila *itens; /* Ponteiro para o array que conterá os itens */
   int        frente, fundo;
   int        tamanhoArray; /* Número de elementos do array */
};
```

Quando os elementos de uma fila são armazenados num array dinâmico, implementá-la como circular deixa de ser vantajoso. Isso ocorre porque, após um redimensionamento do array que armazena os elementos da fila, as funções que utilizam o tamanho do array em suas definições deixam de fazer sentido (v. **Seção 8.3**), a não ser que a fila circular seja rearranjada para adaptar-se ao novo tamanho do array. Além disso, quando uma fila é implementada usando-se um array dinâmico, a operação de enfileiramento tem custo de custo temporal $\theta(n)$ no pior caso, quer a fila seja linear ou circular. Logo, nesse caso, mais uma vez, o uso de fila circular deixa de ser vantajoso.

A seguir serão apresentadas apenas as definições de funções que são substancialmente diferentes daquelas apresentadas na **Seção 8.2**.

```
void CriaFilaIdxD(tFilaIdxD *f)
{
      /* Aloca espaço para a estrutura que armazena a fila */
   *f = malloc(sizeof(struct rotFilaIdxD));

      /* Garante que a alocação realmente ocorreu */
   ASSEGURA(*f, "Nao foi possivel alocar fila");

      /* Inicia a frente e o fundo da fila */
   (*f)->frente = (*f)->fundo = -1;

      /* Aloca um array inicial */
   (*f)->itens = calloc(ACRESCIMO, sizeof(tItemFila));

      /* Garante que o array foi realmente alocado */
   ASSEGURA( (*f)->itens, "Nao foi possivel alocar "
             "array para conter elementos da fila" );

      /* Atualiza a variável que armazena o tamanho do array */
   (*f)->tamanhoArray = ACRESCIMO;
}

void DestroiFilaIdxD(tFilaIdxD *fila)
{
      /* Se a fila não estiver vazia, libera o espaço */
      /* ocupado pelo array que contém os elementos    */
   if (!EstaVaziaFilaIdxD(fila))
      free((*fila)->itens);

      /* Libera a estrutura que representa a fila */
   free((*fila));
}

void EnfileiraIdxD(tFilaIdxD *f, const tItemFila *item)
{
   tItemFila *novoArray; /* Ponteiro para o array redimensionado */

      /* Se o array estiver repleto, tenta redimensioná-lo */
   if (EstaCheiaFilaIdxD(f)) {
         /* Calcula o novo tamanho do array */
      (*f)->tamanhoArray += ACRESCIMO;

         /* Tenta redimensionar o array */
```

```
      novoArray = realloc((*f)->itens, (*f)->tamanhoArray*sizeof(tItemFila));

         /* Checa se houve realocação de memória */
      ASSEGURA(novoArray, "Nao houve redimensionamento");

         /* O ponteiro que apontava para o início do array pode  */
         /* não ser mais válido. Portanto é preciso atualizá-lo. */
      (*f)->itens = novoArray;
   }

   ++(*f)->fundo;
   (*f)->itens[(*f)->fundo] = *item;
}
```

9.8 Análise de Implementações com Arrays Dinâmicos

As operações sobre listas indexadas implementadas utilizando arrays dinâmicos são afetadas pelo uso de **realloc()** para redimensionamento de um array que armazena uma lista. Essa função tem custos espacial e temporal $\theta(n)$ no pior caso. Portanto operações que não usam essa função apresentam o mesmo custo apresentada para listas implementadas como arrays estáticos (v. **Seção 7.3**).

Todas as operações sobre listas indexadas implementadas usando arrays dinâmicos por meio das seguintes funções apresentam custo temporal $\theta(1)$ no pior caso, pois essas operações independem do tamanho da entrada (comprimento da lista):

- `CriaListaIdxD()`
- `ComprimentoListaIdxD()`
- `ObtemElementoListaIdxD()`
- `EstaVaziaListaIdxD()`
- `AlteraElementoListaIdxD()`

As funções `InsereListaIdxD()`, `InsereEmOrdemIdxD2()` e `AcrescentaListaIdxD()` apresentam custo temporal $\theta(1)$ no melhor caso, que ocorre quando a inserção ocorre ao final da lista e não há redimensionamento do array que armazena a lista. Essas funções apresentam custo temporal $\theta(n)$ nos casos pior e mediano com ou sem redimensionamento do array. O pior caso ocorre quando a inserção se dá no início da lista ou quando a inserção ocorre ao final e há redimensionamento do array. O caso mediano ocorre quando a inserção é no meio da lista.

Quando o array que armazena a lista não é redimensionado, a função `RemoveListaIdxD()` apresenta custo temporal o $\theta(1)$ no melhor caso e $\theta(n)$ nos casos pior e mediano (v. **Seção 7.3**). Quando há redimensionamento do referido array, o custo temporal dessa função é $\theta(1)$ no melhor caso e $\theta(n)$ nos demais casos.

Com exceção da operações que requerem redimensionamento de array, que têm custo espacial $\theta(n)$, todas as demais operações sobre listas indexadas discutidas neste capítulo têm custo espacial $\theta(1)$, pois nenhuma delas requer espaço adicional que dependa do tamanho da lista.

Utilizando a abordagem apresentada na **Seção 9.7**, as únicas operações sobre pilhas e filas que não apresentam custo temporal igual a $\theta(1)$ são as operações de empilhamento e enfileiramento. Essas duas operações têm custo $\theta(n)$ no pior caso, que é aquele que requer redimensionamento do array utilizado.

9.9 Exemplos de Programação

9.9.1 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):

```
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 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;

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

         /* 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 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 apontado 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;
}
```

Antes de tentar entender o funcionamento da função `LeLinhaIlimitada()`, que é relativamente complexa, é essencial que você assimile bem o que essa função exatamente faz. Com esse propósito, observe na **Tabela 9–2** a comparação entre essa função e a função **fgets()**, que faz parte da biblioteca padrão de C (`#include <stdio.h>`).

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 9–2: COMPARANDO fgets() E `LeLinhaIlimitada()`

A função `LeLinhaIlimitada()` possui dois parâmetros:

- `tam` (saída) — se não for **NULL**, esse parâmetro apontará para uma variável que armazenará o tamanho do string constituído pelos caracteres da linha lida. Esse parâmetro pode ser **NULL** e, nesse caso, o tamanho do string não será armazenado.
- `stream` (entrada) — stream de texto no qual será feita a leitura. Esse stream deve estar associado a um arquivo de texto aberto em modo de texto que permita leitura.

Quando é bem-sucedida, a função `LeLinhaIlimitada()` retorna o endereço do array contendo a linha lida. Se ocorrer erro ou o final do arquivo for atingido antes da leitura de qualquer caractere, essa função retorna **NULL**.

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, é preciso 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 malsucedida.

```
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 nele 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 9.2**, 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 9.2**).

A próxima instrução no corpo do laço testa se a alocação foi malsucedida e, se esse for o caso, libera o espaço alocado anteriormente para o array e retorna **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 executadas 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 despercebida. De fato, 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): A função **main()** apresentada a seguir lê linhas de tamanho arbitrário num arquivo de texto e via teclado, e apresenta-as na tela.

```
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);

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

         /* 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;
}
```

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 9.2). Essa função **main()** faz uso das seguintes constantes simbólicas:

```
   /* 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
```

9.9.2 A Urupema de Eratóstenes

Preâmbulo: A **Urupema de Eratóstenes**[1] 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.

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

Problema: (a) 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): A função `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**. O parâmetro `n` (entrada) representa o maior valor que essa função verificará se é primo. Essa função retorna `1`, se ocorrer erro ou `0`, se não ocorrer erro.

```
int Eratostenes(int n)
{
   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 0; */
      /* 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 */
}
```

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 dedicado 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): A função **main()** apresentada a seguir 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.

```
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 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;
}
```

Um programa contendo as funções descritas acima deve incluir a seguinte definição de constante simbólica:

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

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
```

9.9.3 Concatenação Múltipla de Strings

Problema: (a) Escreva uma função que concatena um string um número especificado de vezes. (b) Escreva um programa para testar a função especificada no item (a).

Solução de (a): A função `ConcatenaNVezes()` definida abaixo cria um array dinamicamente e armazena nele o resultado da concatenação do string recebido como parâmetro. Seus parâmetros são: `n` (entrada), que especifica o número de concatenações e `str` (entrada), que é o string que será concatenado `n` vezes. Essa função retorna o endereço de um array alocado dinamicamente contendo o resultado da concatenação, se ela for bem-sucedida, ou **NULL**, em caso contrário.

```
char *ConcatenaNVezes(int n, const char *str)
{
   char *concatenacao; /* Apontará para o resultado */
   int   i;
      /* O valor de n deve ser pelo menos igual a 1 */
   if (n <= 0)
      return NULL; /* Por que esta função foi chamada? */

      /* Tenta alocar o espaço necessário para conter o resultado da concatenação. */
      /* É necessário adicionar 1 devido ao onipresente caractere '\0'             */
   concatenacao = malloc(n*strlen(str) + 1);

      /* Se o espaço não foi alocado, retorna-se NULL */
   if (!concatenacao)
      return NULL; /* Alocação falhou */

      /* Primeiro, copia o string para o array alocado */
   strcpy(concatenacao, str);

      /* Agora, acrescenta, sucessivamente, n-1 cópias do string ao mesmo array */
   for (i = 1; i < n; i++)
      strcat(concatenacao, str); /* Acrescenta o string mais uma vez */

   return concatenacao;
}
```

Solução de (b): A função **main()** definida a seguir concatena um string especificado pelo usuário o número de vezes escolhido por ele.

```
int main(void)
{
   char *string, /* String que será concatenado n vezes */
        *resultado; /* Resultado da concatenação */
   int   nVezes; /* Número de vezes que o string será concatenado */
      /* Apresenta o programa */
   printf( "\n\t>>> Este programa une os caracteres que"
           "\n\t>>> voce especificar o numero de vezes"
           "\n\t>>> que voce desejar, e apresenta o"
           "\n\t>>> resultado na tela.\n" );
      /* Lê o string que o usuário deseja concatenar */
   printf("\n\t>>> Que caracteres deseja unir?\n\t> ");
   string = LeLinhaIlimitada(NULL, stdin);
      /* Se o string lido for nulo, não há o que fazer */
   if (!*string) {
      printf("\n\t>>> Nao li nenhum caractere. Tchau.\n");
      return 1;
   }
      /* Solicita o número de concatenações ao usuário */
   printf( "\n\t>>> Quantas vezes deseja unir os "
           "caracteres? " );
      /* Se o número de concatenações não for aceitável, encerra o programa */
   if ((nVezes = LeInteiro()) <= 0) {
      printf("\n\t>>> O valor deveria ser pelo menos igual a 1. Tchau.\n");
```

```
      return 1;
   }
      /* Obtém o resultado da concatenação */
   resultado = ConcatenaNVezes( nVezes, string );
      /* Apresenta o resultado da concatenação */
   if (resultado) {
      printf("\n>>> Resultado da concatenacao de \"%s\" %d vezes:", string, nVezes);
      printf("\n\n%s\n", resultado);
   } else
      printf("\n>>> Nao foi possivel concatenar \"%s\" %d vezes:", string, nVezes);

      /* Libera os espaços alocados dinamicamente. Defato, isso não é necessário, */
      /* já que o programa encerrará em seguida, mas para não perder o hábito...  */
   free(string);
   free(resultado);
   return 0;
}
```

A função **main()** chama a função `LeLinhaIlimitada()`, apresentada na **Seção 9.9.1**, para ler o string introduzido pelo usuário. Essa função **main()** utiliza a seguinte constante simbólica definida no respectivo programa:

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

Exemplo de execução do programa:

```
        >>> Este programa une os caracteres que
        >>> voce especificar o numero de vezes
        >>> que voce desejar, e apresenta o
        >>> resultado na tela.
        >>> Que caracteres deseja unir?
        > +-
        >>> Quantas vezes deseja unir os caracteres? 20

>>> Resultado da concatenacao de "+-" 20 vezes:
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
```

Apesar de esse exemplo parecer um mero exercício didático, ele tem um aspecto prático, que é exatamente desenhar linhas com os caracteres e o tamanho especificados, como mostra o exemplo de execução do programa.

9.10 Exercícios de Revisão

Motivação e Justificativas (Seção 9.1)

1. Cite três situações em programação nas quais alocação dinâmica de memória se faz necessária.
2. Defina (a) variável estática e (b) variável dinâmica.
3. (a) O que é alocação estática de memória? (b) O que é alocação dinâmica de memória?
4. (a) O que é subdimensionamento de memória? (b) O que pode ocorrer com um programa que apresenta subdimensionamento de memória?
5. (a) O que é superdimensionamento de memória? (b) O que ocorre com um programa que apresenta superdimensionamento de memória?

Funções de Alocação e Liberação Dinâmica de Memória (Seção 9.2)

6. 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));
```

7. (a) Variáveis definidas com **static** são consideradas variáveis estáticas? (b) E quanto a variáveis definidas sem **static**?
8. Por que ponteiros devem ser iniciados com **NULL** na ausência de um valor mais específico?
9. Descreva o funcionamento de cada uma das funções a seguir:
   (a) **malloc()**
   (b) **calloc()**
   (c) **free()**
10. (a) Quando o valor retornado por **realloc()** é **NULL**, o ponteiro passado como primeiro parâmetro para essa função continua válido? (b) Quando o valor retornado por **realloc()** é diferente de **NULL**, o ponteiro passado como primeiro parâmetro para essa função continua válido?
11. Como funciona a função **realloc()** quando o primeiro parâmetro é **NULL**?
12. Como funciona a função **realloc()** quando o valor do segundo parâmetro é zero?
13. Por que não é aconselhável atribuir o valor retornado por **realloc()** ao mesmo ponteiro passado como primeiro parâmetro numa chamada dessa função?
14. (a) O que é uma variável anônima? (b) Por que variáveis anônimas aparecem apenas no contexto de alocação dinâmica de memória?
15. Variáveis estáticas são liberadas automaticamente (v. **Seção 2.3**). Então, por que variáveis dinâmicas precisam ser liberadas explicitamente?
16. Descreva dois erros comuns de liberação de blocos alocados dinamicamente e como o programador pode precaver-se contra eles.
17. A função `CriaArray()` a seguir foi implementada com a intenção de alocar dinamicamente um array de elementos do tipo **int**. Essa função chama `LeInteiro()` definida na **Seção 2.7.1** 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;
}
```

18. Suponha que `p` seja um ponteiro que 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));
```

Ponteiros Genéricos e o Tipo void * (Seção 9.3)

19. (a) O que é um ponteiro genérico? (b) Em que situações ponteiros genéricos são úteis?

20. Como ponteiros genéricos são definidos?

21. Em quais situações o tipo **void** * é normalmente utilizado?

22. Apresente dois exemplos de uso do tipo **void** *.

23. (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;
}
```

A Partição de Memória Heap (Seção 9.4)

24. Como tipicamente é dividido o espaço reservado para a execução de um programa?

25. Qual é a importância de heap em alocação dinâmica de memória?

26. (a) O que significa fragmentação de heap? (b) Qual pode ser a consequência danosa decorrente de fragmentação de heap?

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

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

Testando Alocação Dinâmica de Memória (Seção 9.5)

29. (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?

30. Como a macro `ASSEGURA`, definida na **Seção 7.4**, pode ser usada para garantir que uma chamada de função de alocação dinâmica de memória foi bem-sucedida?

31. 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;
}
```

Listas Indexadas Dinâmicas (Seção 9.6)

32. (a) O que é um array estático? (b) O que é um array dinâmico?

33. Qual é a principal vantagem obtida com o uso de arrays dinâmicos em relação ao uso de arrays estáticos?

34. Qual é o papel desempenhado por **realloc()** no processamento de listas dinâmicas?

35. Discuta as abordagens básicas de implementação de listas usando arrays dinâmicos discutidas na **Seção 9.6** apresentando as vantagens e desvantagens de cada uma delas.

36. (a) Para que serve a função `DestroiListaIdxD()` apresentada na **Seção 9.6**? (b) Se os TADs discutidos na **Seção 9.6** fossem implementados por meio de arrays estáticos, em vez de arrays dinâmicos, essa função seria necessária?

37. Uma operação de busca numa lista indexada depende do fato de essa lista ser implementada como array estático ou dinâmico?

38. Por que o ponteiro `lista->elementos` utilizado como parâmetro na chamada de **realloc()** no corpo da função `AcrescentaListaIdxD()` pode deixar de ser válido quando a função **realloc()** é bem-sucedida?

Pilhas e Filas Implementadas com Arrays Dinâmicos (Seção 9.7)

39. Os protótipos das funções apresentadas na **Seção 8.1** e os protótipos das funções correspondentes apresentadas na **Seção 9.7.1** são iguais. Explique a importância desse fato.

40. Por que a função `CriaPilhaIdxD()` definida na **Seção 9.7.1** efetua uma chamada de **malloc()** e outra de **calloc()**?

41. Por que a função `CriaPilhaIdxD()` definida na **Seção 9.7.1** efetua duas chamadas de **free()**?

42. Por que uma fila circular não faz sentido quando os itens da fila são armazenados em arrays dinâmicos?

Análise de Implementações com Arrays Dinâmicos (Seção 9.8)

43. (a) Por que o custo temporal da função **realloc()** é $\theta(n)$ no pior caso? (b) Por que o custo espacial da função **realloc()** é $\theta(n)$ no pior caso?

44. Por que a função `AcrescentaListaIdxD()` apresentada na **Seção 9.6** tem custo temporal $\theta(n)$ enquanto a função `AcrescentaListaIdx()` definida na **Seção 7.1** tem custo temporal $\theta(1)$?

45. Considere o seguinte raciocínio:

> A função `RemoveListaIdxD()` teria custo temporal $\theta(n)$ no pior caso se ela não chamasse **realloc()**. Assim, como a função **realloc()** tem custo temporal $\theta(n)$ no pior caso, aplicando-se a regra da soma (**Teorema 6.4**), tem-se que o custo temporal da função `RemoveListaIdxD()` deve ser $\theta(2n)$.

Qual é o equívoco desse raciocínio?

46. Qual é o custo temporal da função `DestroiListaIdxD()`?

Exemplos de Programação (Seção 9.9)

47. Qual é a utilidade da função `LeLinhaIlimitada()` definida na **Seção 9.9.1**?

48. Quais são as diferenças e semelhanças entre as funções `LeLinhaIlimitada()`, definida na **Seção 9.9.1**, e a função **fgets()** da biblioteca padrão de C?

49. Explique o algoritmo conhecido como Crivo de Eratóstenes.

9.11 Exercícios de Programação

EP9.1 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*);
```

EP9.2 (a) Escreva uma função que crie 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).

EP9.3 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 (i.e., `"r"`). (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 as funções **fgetc()** e **putchar()**.]

EP9.4 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.

EP9.5 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()` 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.]

EP9.6 Escreva um TAD que defina o tipo `tString`. Esse TAD deve incluir operações para:

(a) Criar um string
(b) Calcular o comprimento de um string
(c) Clonar um string
(d) Concatenar dois strings
(e) Comparar dois strings

EP9.7 (a) Escreva uma função que remova todas as ocorrências de um string de um stream de texto e armazene 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 9.9.1**, 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
```

EP9.8 (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 9.9.1**, 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"
```

EP9.9 Considerando que o arquivo binário `Tudor.bin`, criado pelo programa da **Seção 7.6.2**, armazena dados de uma turma escolar. Escreva um programa que faça o seguinte:

(a) Lê estruturas do tipo `tAluno` no arquivo `Tudor.bin` e armazena-as num array dinâmico ordenado por um campo de estrutura especificado pelo usuário por:

```
1. Nome
2. Matrícula
3. Primeira nota
4. Segunda nota
5. Média
6. Sem ordenação
```

(b) Apresenta um menu com as seguintes opções para o usuário:

```
A. Acrescenta um aluno
R. Remove um aluno
E. Exibe turma na tela
C. Consulta dados de aluno
L. Altera dados de aluno
T. Altera a ordenação da turma
I. Inverte a ordenação da turma
N. Encerra o programa
```

(c) Enquanto o usuário não escolher a opção de saída do programa, lê a opção escolhida pelo usuário e executa a operação correspondente.

(d) Logo antes de encerrar, se houve alteração de dados durante a execução do programa, o arquivo utilizado deve ser atualizado.