Capítulo 9

Caracteres e Strings

Após estudar este capítulo, você deverá ser capaz de:

- Definir e usar os seguintes conceitos:
  - ❒ String
  - ❒ String constante
  - ❒ argc
  - ❒ Argumento
  - ❒ Caractere nulo
  - ❒ Classificação de caractere
  - ❒ argv
- Descrever o funcionamento das seguintes funções da biblioteca padrão de C:
  - ❒ **strlen()**
  - ❒ **strstr()**
  - ❒ **strcat()**
  - ❒ **isalpha()**
  - ❒ **isspace()**
  - ❒ **strcpy()**
  - ❒ **strchr()**
  - ❒ **isalnum()**
  - ❒ **isdigit()**
  - ❒ **isupper()**
  - ❒ **strcmp()**
  - ❒ **strrchr()**
  - ❒ **isalpha()**
  - ❒ **islower()**
  - ❒ **atoi()**
  - ❒ **strcoll()**
  - ❒ **strtok()**
  - ❒ **isblank()**
  - ❒ **ispunct()**
  - ❒ **strtod()**
- Usar as principais funções de processamento de strings da biblioteca padrão de C
- Implementar um programa que recebe entrada por meio de argumentos
- Explicar por que nem todo array de caracteres é um string
- Discorrer sobre a importância de strings em programação
- Expor por que um programa pode ser abortado ao tentar alterar o conteúdo de um string constante
- Discutir o uso preventivo de **const** na definição de ponteiros para strings constantes
- Explicar por que raramente o operador **sizeof** pode ser usado para determinar o tamanho de um string
- Descrever as facilidades oferecidas pela função `LeString()`
- Saber que, em nenhum padrão de C, **main()** tem tipo de retorno **void**
- Saber como um programa pode obter seu nome de arquivo executável
- Explicar por que não se devem fazer suposições sobre a ordem de avaliação de parâmetros numa chamada de função

OBJETIVOS

9.1 Introdução

EM C, UM STRING é um array de elementos do tipo **char** terminado pelo **caractere nulo**, representado pela sequência de escape `'\0'`. Em qualquer código de caracteres usado numa implementação de C, o inteiro associado a esse caractere é zero.

Strings constituem um tipo de dado essencial para qualquer programa interativo, pois mesmo programas que possuem interfaces gráficas sofisticadas precisam ler, processar e exibir strings. Por isso, a maioria das linguagens de programação de alto nível oferece muitas operações de manipulação de strings prontas para uso. Em C, essas operações são implementadas como funções que fazem parte do módulo string da biblioteca padrão da linguagem. As funções de processamento de strings da biblioteca padrão de C mais comumente utilizadas serão apresentadas neste capítulo.

Este capítulo também discute as funções do módulo de biblioteca ctype que contém funções dedicadas à classificação e transformação de caracteres.

9.2 Armazenamento de Strings em Arrays de Caracteres

Pode-se armazenar um string em memória utilizando-se um array de elementos do tipo **char** iniciado como mostra o seguinte exemplo:

```
char  ar1[] = "bola";
```

Devido à onipresença do caractere terminal de string `'\0'`, quando não é especificado explicitamente, o tamanho de um array iniciado com um string é sempre um a mais do que o número de caracteres visíveis no string. Assim, o array `ar1[]` do exemplo acima possui tamanho igual a `5` (i.e., quatro caracteres visíveis mais o caractere terminal).

Na realidade, apesar da aparência, os caracteres entre aspas que aparecem numa iniciação de um array de caracteres não constituem um string constante. Isto é, na **Seção 3.5.4**, um string constante foi definido como uma sequência de caracteres entre aspas, mas essa definição não vale nesse caso específico de iniciação de array. Quer dizer, nesse caso, caracteres entre aspas constituem uma facilidade oferecida pela linguagem C para isentar os programadores de ter que escrever iniciações de arrays de caracteres do modo convencional (v. **Seção 8.4**), que é bem mais trabalhoso. Ou seja, usando a notação convencional de iniciação de arrays, o array `ar1[]` do último exemplo deveria ser escrita assim:

```
char  ar1[] = {'b', 'o', 'l', 'a', '\0'};
```

Quando o número de caracteres numa iniciação de array é relativamente grande, é permitido dividi-los em partes menores separadas por quebras de linha, tal qual ocorre com strings constantes. O seguinte programa ilustra essa facilidade:

```
#include <stdio.h>

int main(void)
{
   char domPedroI[] = "Pedro de Alcantara Francisco Antonio Joao Carlos Xavier de "
                      "Paula Miguel  Rafael Joaquim Jose Gonzaga Pascoal Cipriano "
                      "Serafim de Braganca e Bourbon";

   printf("\nNome completo de D. Pedro I: %s\n", domPedroI);

   return 0;
}
```

Já imaginou se você tivesse que iniciar um array com o nome de D. Pedro I se não existissem essas facilidades?

Deve-se notar que, quando o número de elementos do array é especificado e é menor do que ou igual ao número de caracteres presentes na iniciação (v. **Seção 8.4**), o array não conterá um string. Por exemplo, na seguinte iniciação, o array `ar2[]` receberá apenas os caracteres: `'b'`, `'o'`, `'l'` e `'a'` e, portanto, não conterá um string em virtude da ausência do caractere terminal `'\0'`.

```
char ar2[4] = "bola";
```

Quando o número de elementos do array é especificado e é maior do que o número de caracteres presentes na iniciação, os elementos remanescentes no array, se for o caso, serão iniciados com zero. Por exemplo, a definição abaixo é equivalente àquela do array `ar1[]` no início desta seção.

```
char ar3[5] = "bola";
```

Por outro lado, na iniciação do array `ar4[]` abaixo:

```
char ar4[10] = "bola";
```

`ar4[0]` recebe o valor `'b'`, `ar4[1]` recebe `'o'`, `ar4[2]` recebe `'l'`, `ar4[3]` recebe `'a'` e `ar4[4]` assume o valor `'\0'`. Os elementos restantes (i.e., de `ar4[5]` a `ar4[9]`) recebem zero como valor.

9.3 Strings Constantes

Um **string constante** é uma sequência de caracteres entre aspas, desde que tal construção não apareça na iniciação de um array de caracteres, conforme foi visto na **Seção 9.2**. Um string constante é representado pelo endereço de seu primeiro caractere em memória. Portanto o tipo de um string constante é **char** * e pode-se iniciar um ponteiro para **char** com um string constante, como, por exemplo:

```
char *ptr = "Isto e' um string constante."
```

Entretanto, essa definição de variável é diferente das definições apresentadas na **Seção 9.2** que utilizam arrays. Uma diferença entre essa última definição e a definição:

```
char str[] = "Isto NAO e' um string constante."
```

é que, no primeiro caso, além do espaço reservado para conter o string, também é alocado espaço para conter o ponteiro `ptr`. Além disso, apesar de `ptr` e `str` apontarem para o elemento inicial do string (i.e., para o caractere `'I'`), o valor da variável `ptr` pode ser modificado, enquanto o endereço `str` não pode (v. **Seção 8.7**). Entretanto, se o valor de `ptr` for modificado, o endereço com o qual esse ponteiro foi iniciado será perdido (i.e., o string para o qual `ptr` estava apontando não poderá mais ser acessado).

Uma importante diferença entre as iniciações do array `str[]` e do ponteiro `ptr` acima é o fato de strings constantes poderem ser armazenados numa região de memória cujo conteúdo não pode ser modificado (i.e., os bytes nessa região são apenas para leitura). Em tal situação, qualquer tentativa de modificar o string para o qual o ponteiro `ptr` aponta gera um erro de execução (i.e., aborto) do programa. Para não correr riscos, considere como *realmente constantes* os conteúdos de strings constantes cujos endereços são atribuídos a ponteiros.

Para precaver-se contra possíveis problemas causados por alterações indevidas de strings constantes use **const** na definição de ponteiros para strings constantes como mostrado abaixo:

```
const char *ptr = "Isto e' um string constante."
```

9.4 Comparando Ponteiros, Strings e Caracteres

As diferenças entre strings constantes contendo apenas um caractere visível e caracteres constantes constituem uma fonte comum de confusão entre iniciantes em C. Uma dessas diferenças refere-se ao espaço ocupado por um caractere constante e um string constante consistindo de apenas um caractere: no primeiro caso, apenas

um byte é alocado em memória, enquanto, no segundo caso, dois bytes são alocados em virtude do caractere nulo de terminação do string. Por exemplo, o caractere constante `'A'` ocupa apenas um byte, enquanto o string constante `"A"` ocupa dois bytes. Essas diferenças são resumidas na Tabela 9–1.

	Caractere Constante	String Constante com um Caractere
Espaço ocupado	1 byte	2 bytes
Tipo	**char**	**char** *
Exemplo	`'A'`	`"A"`

Tabela 9–1: Caractere Constante versus String Constante

Considere, no presente contexto, uma instrução *legal* como sendo aquela que não contraria as regras da linguagem C e, portanto, pode ser compilada. Por outro lado, considere uma instrução *anormal* como sendo legal, mas desprovida de significado prático ou que cause um sério problema para o programa que a contenha (nesse caso, um bom compilador emite mensagens de advertência). Então, é normal atribuir um caractere constante ao conteúdo apontado por um ponteiro para o tipo **char**, mas o mesmo não é verdade com respeito a um string constante. Por exemplo, se `p` é um ponteiro para **char**, a atribuição:

```
*p = 'a';  /* Legal e normal */
```

é perfeitamente legal, mas a atribuição abaixo, apesar de legal, não faz sentido:

```
*p = "a";  /* Legal mas não é normal */
```

Como um string constante é interpretado como o endereço de seu primeiro caractere e o resultado da indireção de um ponteiro é do mesmo tipo do valor para o qual o ponteiro aponta, essa última instrução tenta atribuir um endereço a uma variável do tipo **char**.

É legal e normal atribuir um string a um ponteiro para **char**, mas, certamente, será problemático atribuir um caractere constante a um ponteiro. Por exemplo, se `p` é um ponteiro para **char**, a primeira instrução a seguir é legal e perfeitamente apropriada, mas a segunda, apesar de legal, provavelmente trará dissabores quando o programa que a contém for executado.

```
p = "a";   /* Legal e normal */
p = 'a';   /* Problema à vista! */
```

A Tabela 9–2 resume os exemplos apresentados nesta seção.

Exemplo	Normal?	Justificativa
`*p = 'a'`	Sim	Os dois lados da atribuição são do tipo **char**
`*p = "a"`	Não	O lado esquerdo da atribuição é do tipo **char** e o lado direito é do tipo **char** *
`p = "a"`	Sim	Os dois lados da atribuição são do tipo **char** *
`p = 'a'`	Não	O lado esquerdo da atribuição é do tipo **char** * e o lado direito é do tipo **char**

Tabela 9–2: Atribuições entre Ponteiros, Caracteres e Strings

Uma confusão comum entre iniciantes em C é imaginar que iniciações e atribuições de ponteiros são equivalentes. Por exemplo, a iniciação:

```
char *p = "string"; /* O asterisco aqui é definidor */
```

é legal e absolutamente correta, pois se está atribuindo um endereço a um ponteiro. Isto é, a atribuição é feita a `p` e não a `*p`, pois o asterisco aqui é usado como definidor de ponteiro e não como operador de indireção.

Entretanto, a instrução:

```
*p = "string"; /* O asterisco aqui é operador de indireção */
```

é problemática, pois se está tentando atribuir um endereço a uma variável do tipo **char** (i.e., `*p`).

Considerando as seguintes definições de variáveis:

```
char  ar[10];
char  *ptr = "10 espacos";
```

os exemplos apresentados a seguir ajudam a esclarecer algumas dúvidas que um programador inexperiente pode ter com o uso de ponteiros, arrays e strings.

```
ar = "errado";   /* ILEGAL */
```

Essa instrução é *ilegal* porque `ar` representa o endereço do elemento inicial do array `ar[]` e esse endereço não pode ser modificado.

```
ar[2] = 'a';
```

Essa instrução é *legal*: ela representa a atribuição do caractere `'a'` ao terceiro elemento do array `ar[]`.

```
ptr[5] = 'b';
```

Essa instrução é *legal* por causa da relação entre ponteiros e arrays (v. **Seção 8.7**). Essa atribuição modifica o valor do elemento de índice `5` do string `"10 espacos"` para `'b'`, de modo que esse string se torna `"10 esbacos"`. Ademais, o valor do ponteiro `ptr` em si *não é modificado*. Mas, apesar de ser legal, essa instrução pode causar aborto de programa se o string for armazenado numa área de memória reservada apenas para leitura. Portanto é melhor evitar instruções que tentem modificar strings considerados constantes.

```
*(ptr + 5) = 'b';
```

Essa instrução é *legal*, é exatamente equivalente à instrução anterior e pode causar o mesmo problema que aquela instrução.

```
ptr = "OK";
```

Essa instrução é obviamente legal. Talvez, menos óbvio seja o fato de `ptr` agora apontar para outra posição de memória, que é aquela na qual o string `"OK"` é armazenado. O endereço do string antigo para onde `ptr` apontava ficará perdido e aquele string não poderá mais ser acessado.

```
ptr[5] = 'b';
```

Essa instrução é sintaticamente legal, mas irá provavelmente causar dissabores quando for executada. O problema é que, em consequência da atribuição no item anterior, `ptr` aponta agora para o string `"OK"`, que possui apenas três bytes alocados e essa última instrução atribui o valor `'b'` ao terceiro byte além do final do string `"OK"`. Portanto essa instrução tenta modificar um espaço em memória que não está alocado para o referido string. Em outras palavras, essa instrução não causará um erro de compilação, mas certamente causará um erro durante a execução do programa.

```
*ptr = "ilegal?";
```

Essa instrução não é ilegal, mas incorpora dois problemas. Primeiro, ela é desprovida de significado porque está sendo atribuído ao caractere para o qual `ptr` aponta o endereço do elemento inicial do string constante

`"ilegal?"`. Logo essa atribuição representa, na realidade, a tentativa de atribuição de um endereço a uma variável do tipo **char**. Nesse caso, o padrão ISO requer apenas que o compilador emita uma mensagem de advertência. O segundo problema é pior: como no instante da execução dessa instrução o ponteiro `ptr` aponta para o string constante `"OK"`, provavelmente o programa será abortado, como ocorre nos outros casos de tentativa de alteração de strings constantes descritos antes.

```
printf("%c", 3["Estranho"]);
```

Por mais estranho que possa parecer, essa instrução é perfeitamente legal e escreve o caractere `'r'` na tela. Para entender por que a expressão `3["Estranho"]` é legal, note que, de acordo com a relação entre ponteiros e arrays (v. Seção 8.7), essa expressão é equivalente a `3 + "Estranho"`, já que o string constante `"Estranho"` é avaliado como o endereço do local onde ele é armazenado, conforme foi visto na Seção 9.3. Ora, mas a soma é uma operação comutativa; portanto, `3 + "Estranho"` é o mesmo que `"Estranho" + 3` e essa última expressão é, novamente usando a relação apresentada na Seção 8.7, o mesmo que `"Estranho"[3]`. Essa última expressão é evidentemente legal e representa o caractere de índice `3` do string `"Estranho"`.

É muito importante que você entenda todos os exemplos apresentados nesta seção antes de prosseguir.

9.5 Funções de Biblioteca para Processamento de Strings

A biblioteca padrão de C possui várias funções para processamento de strings declaradas no cabeçalho `<string.h>`. Por outro lado, leitura e escrita de strings utilizam funções declaradas em `<stdio.h>` (v. Seção 3.13.1). Mas, para facilitar a tarefa de leitura de strings, por enquanto, será utilizada a função `LeString()` da biblioteca LeituraFacil.

9.5.1 Leitura de Strings via Teclado: LeString()

A função `LeString()` faz parte da biblioteca LeituraFacil e tem o seguinte protótipo:

```
int LeString(char *ar, int nElementos)
```

O parâmetro `ar` representa o endereço de um array com tamanho suficiente para conter todos os caracteres que se pretendem ler e `nElementos` é o número de elementos do array mencionado. A função `LeString()` lê e armazena caracteres no array apontado por `ar` até encontrar uma quebra de linha (`'\n'`) ou atingir o máximo de `nElementos - 1` caracteres lidos. Quando lido, o caractere `'\n'` não é incluído no array, mas o caractere `'\0'` é acrescentado após o último caractere armazenado no array, de modo que, ao final da operação, o array sempre conterá um string.

Quando o usuário digita apenas `[ENTER]`, a função `LeString()` armazena no array supracitado um **string vazio**; i.e., contendo apenas o caractere terminal.

Pode-se testar se um string vazio foi armazenado em um array usando uma instrução **if** conforme mostra o seguinte trecho de programa:

```
char str[20];

printf("\nDigite uma sequencia de caracteres: ");
LeString(str, 20);

if (*str == '\0') { /* String é vazio */
   printf("Voce nao digitou nenhum caractere\n");
} else { /* String NÃO é vazio */
   printf("Caracteres digitados: %s\n", str);
}
```

A instrução **if** acima pode ser escrita de modo mais compacto como:

```
if (!*str) { /* String é vazio? */
   printf("Voce nao digitou nenhum caractere\n");
} else {
   printf("Caracteres digitados: %s\n", str);
}
```

A função `LeString()` retorna o número de caracteres que o usuário digitou além do que era esperado, se esse for o caso. Um programa pode ignorar esse valor retornado ou usá-lo para decidir como agir, como mostra o exemplo a seguir:

```
#include <stdio.h>   /* printf()   */
#include "leitura.h" /* LeString() */

#define MAX_NOME 30

int main(void)
{
   char  nome[MAX_NOME + 1];
   int   resto;

   printf( "\nDigite seu nome (maximo de %d caracteres)\n\t> ", MAX_NOME );
   resto = LeString(nome, MAX_NOME + 1);

   printf("\nSeu nome e': %s\n", nome);

   if (resto) {
      printf("\a\nVoce digitou %d caracteres excedentes\n", resto);
   }
   return 0;
}
```

Exemplo de execução do programa:

```
Digite seu nome (maximo de 30 caracteres)
        > Pedro de Alcantara Francisco Antonio Joao Carlos Xavier de Paula Miguel
Rafael Joaquim Jose Gonzaga Pascoal Cipriano Serafim de Braganca e Bourbon
Seu nome e': Pedro de Alcantara Francisco A

Voce digitou 116 caracteres excedentes
```

No último exemplo, a atitude do programa quando o usuário digita caracteres a mais do que esperado é apenas emitir uma mensagem de alerta, mas existem alternativas, como mostra o exemplo da Seção 9.10.2.

9.5.2 Escrita de Strings na Tela: printf() e puts()

Strings podem ser escritos na tela utilizando-se a função **printf()** em conjunto com o especificador de formato **%s**. O parâmetro utilizado com **printf()** para escrita de strings na tela deve ser (ou conter) o endereço de um string. A função **printf()** escreve todos os caracteres do referido string até que o caractere nulo seja encontrado. Por exemplo:

```
#include <stdio.h>

int main(void)
{
   char  str[] = "Um string",
        *p = "Outro string";

   printf( "\n>>> Strings:\n\t%s\n\t%s\n\t%s\n", str, p, "Mais um string" );

   return 0;
}
```

Esse programa causaria a escrita do seguinte na tela:

```
>>> Strings:
        Um string
        Outro string
        Mais um string
```

Existe ainda a função **puts()** para escrita de strings na tela, mas a única facilidade adicional oferecida por essa função com relação a **printf()** é que ela provoca uma quebra automática de linha ao final da escrita de um string. Por exemplo, a chamada de **puts()**:

```
puts("Um string");
```

teria o mesmo efeito que a chamada de **printf()**:

```
printf("Um string\n");
```

Note que a função **puts()** não precisa do caractere `'\n'` para provocar quebra de linha, como ocorre com a função **printf()**. Assim como **printf()**, **puts()** faz parte do módulo stdio da biblioteca padrão de C (v. **Seção 3.14**).

9.5.3 Comprimento de Strings: strlen()

A função **strlen()** retorna o comprimento do string que recebe como parâmetro e tem o seguinte protótipo:

```
size_t strlen(const char *string)
```

Um fato interessante sobre o protótipo da função **strlen()** é que ele deveria utilizar a notação:

```
size_t strlen(const char string[])
```

visto que o parâmetro `string` representa um ponteiro para um string (que é um array) e não um ponteiro para um único caractere. Entretanto, no caso de strings, a recomendação apresentada na **Seção 8.9.1** pode ser relaxada, pois é raro se ter um ponteiro que aponta para um único caractere. Isto é, tipicamente um ponteiro do tipo **char** * aponta para o início de um string, e não para um caractere único. É interessante notar ainda o uso de **const** pela função **strlen()** para assegurar que o string recebido como parâmetro não é alterado.

É ainda muito importante observar que o tipo do valor retornado por **strlen()** é **size_t**, que é um tipo inteiro sem sinal (v. **Seção 8.5**). Portanto, para evitar conversão de um valor com sinal num valor sem sinal, não utilize o valor retornado por **strlen()** diretamente numa expressão que não seja de atribuição. Em vez disso, atribua o valor retornado por essa função a uma variável do tipo **int** e, então use essa variável na referida expressão (v. **Seção 8.5**).

A função **strlen()** não inclui o caractere terminal de string `'\0'` na contagem do número de caracteres do string recebido como parâmetro, como mostra a saída do programa a seguir, que apresenta três exemplos de uso da função **strlen()**:

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

int main(void)
{
   char *ptr = "Bola";
   char  ar[] = "Bolada";
   int   tamanho;

   tamanho = strlen(ptr);
   printf("\nTamanho do string \"%s\": %d", ptr, tamanho);
```

```
   tamanho = strlen(ar);
   printf("\nTamanho do string \"%s\": %d", ar, tamanho);

   printf("\nTamanho do string \"Balao\": %d\n", strlen("Balao"));

   return 0;
}
```

Quando executado, esse programa produz a seguinte saída:

```
Tamanho do string "Bola": 4
Tamanho do string "Bolada": 6
Tamanho do string "Balao": 5
```

Nas chamadas de **printf()** do último programa, foram usadas as sequências de escape `'\"'` para exibição de aspas envolvendo os strings.

Apesar de a função **strlen()** existir pronta para uso no módulo string da biblioteca padrão de C, é instrutivo examinar como essa função pode ser implementada em C.

Provavelmente, se solicitado a escrever tal função, um programador iniciante em C executaria essa tarefa como:

```
size_t ComprimentoStr1(const char *str)
{
   size_t tamanho = 0; /* Armazena o tamanho do string  */
   int    i = 0; /* Variável utilizada como índice para */
                 /* acessar cada caractere do string    */

      /* Examina cada caractere do string até     */
      /* encontrar o caractere terminal de string */
   while (str[i] != '\0') {
      ++tamanho; /* Mais um caractere encontrado */
      ++i;
   }

   return tamanho;
}
```

O funcionamento da função `ComprimentoStr1()` é simples e dispensa mais comentários além daqueles encontrados na própria função. Mas, essa função pode ser melhorada notando-se que a variável `i` assume os mesmos valores que a variável `tamanho`. Portanto basta utilizar uma dessas variáveis, em vez de ambas. Escolhendo-se a variável `tamanho` e abandonando-se a variável `i`, pode-se implementar uma nova versão para essa função como:

```
size_t ComprimentoStr2(const char *str)
{
   size_t tamanho = 0; /* Armazena o tamanho do string e usada como     */
                       /* índice para acesso a cada caractere do string */

      /* Examina cada caractere do string até     */
      /* encontrar o caractere terminal de string */
   while (str[tamanho]) {
      ++tamanho; /* Mais um caractere encontrado */
   }

   return tamanho;
}
```

A função `ComprimentoStr2()` utiliza uma variável a menos do que a função `ComprimentoStr1()` e, portanto, é mais eficiente. Outra novidade introduzida na função `ComprimentoStr2()` é que a expressão:

```
str[i] != '\0'
```

utilizada como teste pela instrução **while** na função `ComprimentoStr1()` foi substituída por:

```
str[i]
```

Essas duas expressões são equivalentes porque, em qualquer código de caracteres utilizado em C, o valor de `'\0'` é `0` e, conforme foi visto na Seção 4.5.5, `str[i] != 0` é o mesmo que `str[i]`.

As duas últimas funções apresentadas funcionam perfeitamente bem, mas, provavelmente, um programador de C experiente escreveria uma função que calcula o comprimento de strings de modo mais elegante e sucinto, como mostrado em seguida:

```
size_t ComprimentoStr3(const char *str)
{
   size_t tamanho = 0; /* Armazena o tamanho do string */

      /* Examina cada caractere do string até     */
      /* encontrar o caractere terminal de string */
   while (*str++) {
      ++tamanho; /* Mais um caractere encontrado */
   }

   return tamanho;
}
```

Entender o funcionamento da função `ComprimentoStr3()` requer conhecimento da relação entre ponteiros e arrays unidimensionais (v. Seção 8.7) e das propriedades de precedência e associatividade dos operadores representados por `*` e `++`. Para compreender o funcionamento dessa função, note, em primeiro lugar, que o compilador interpreta o parâmetro `str` como um ponteiro para o primeiro caractere do string cujo comprimento se deseja calcular. Em seguida, observe que o ponto central para compreensão dessa função é a expressão:

```
*str++
```

Essa expressão envolve o uso dos operadores unários representados por `*` e `++`, que fazem parte de um mesmo grupo de precedência. Assim, para saber qual deles é aplicado primeiro, é necessário utilizar a propriedade de associatividade desses operadores que, no caso de todos os operadores unários, é à direita[1]. Logo o primeiro operador a ser aplicado é `++`, *como se a expressão tivesse sido escrita assim*:

```
*(str++)
```

Ora, mas sabe-se que a aplicação do operador sufixo de incremento resulta no próprio operando. Logo, na primeira iteração do laço **while**, o resultado dessa operação é o próprio valor inicial do parâmetro `str`, que é o endereço do primeiro caractere do string. Em seguida, o operador `*` é aplicado, resultando exatamente nesse caractere. Se esse caractere não for nulo, o corpo do laço é executado, resultando no incremento da variável `tamanho`. Na próxima avaliação da expressão `*str++`, o ponteiro `str` apontará para o segundo caractere do string, em virtude do efeito colateral do operador ++ e a história se repete. Assim, o laço **while** encerrará quando o resultado da referida expressão for zero (i.e., quando `str` apontar para o caractere nulo).

Em termos de funcionalidade ou eficiência, não há diferença entre as funções `ComprimentoStr2()` e `ComprimentoStr3()`. Mas, a função `ComprimentoStr3()` é mais elegante e demonstra um conhecimento mais profundo da linguagem C por parte do programador.

9.5.4 Cópia de Strings: strcpy()

A função **strcpy()** copia os caracteres de um string (segundo parâmetro) para um array de caracteres (primeiro parâmetro). Essa função retorna o endereço do array recebido como primeiro parâmetro e seu protótipo é:

[1] Na realidade, rigorosamente falando, de acordo com os padrões ISO C99, e C11 a precedência dos operadores sufixos ++ e -- é maior do que a precedência dos demais operadores unários. Neste livro, por simplicidade, decidiu-se considerar todos os operadores unários como parte de um mesmo grupo de precedência, como foi explicitado no **Prefácio**.

```
char *strcpy(char *destino, const char *origem)
```

Como exemplo de uso dessa função, considere o seguinte programa:

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

int main(void)
{
   char  *ptr = "Bola";
   char   ar[] = "Bolada";
   char   copia[50];
   char  *str;

   str = strcpy(copia, "um string de comprimento 27");
   printf("\nString copiado: \"%s\"", str);
   printf("\nString copiado: \"%s\"", copia);

   strcpy(copia, ptr);
   printf("\nString copiado: \"%s\"", copia);

   strcpy(copia, ar);
   printf("\nString copiado: \"%s\"", copia);

   return 0;
}
```

Como resultado de sua execução, o programa acima exibe o seguinte na tela:

```
String copiado: "um string de comprimento 27"
String copiado: "um string de comprimento 27"
String copiado: "Bola"
String copiado: "Bolada"
```

Na primeira chamada da função **strcpy()** no programa acima:

```
str = strcpy(copia, "um string de comprimento 27");
```

o valor retornado por essa função é atribuído ao ponteiro `str`. As duas chamadas de **printf()** que seguem essa instrução diferem apenas pelo fato de a primeira usar `str` como parâmetro e a segunda usar `copia` como parâmetro. Mas, de acordo com a descrição da função **strcpy()** apresentada, ela retorna o valor recebido como primeiro parâmetro. Portanto a `str` é atribuído o endereço do array `copia[]`, que foi passado como primeiro parâmetro na respectiva chamada de **strcpy()**. Isso explica por que as duas primeiras chamadas de **printf()** apresentam o mesmo string na tela.

Na maioria das vezes, é desnecessário usar o valor retornado por **strcpy()** e isso foi feito aqui por uma razão meramente didática. O valor retornado por essa função é útil quando se deseja usar o resultado da cópia de um string numa outra operação (p. ex., concatenação da cópia de um string com outro string).

É extremamente importante observar que o primeiro parâmetro de **strcpy()** deve apontar para um array com espaço suficiente para conter o string que será copiado (incluindo o caractere terminal). Caso contrário, haverá corrupção de memória (v. **Seção 8.9.3**).

Novamente, é instrutivo que o leitor aprenda como a função **strcpy()** pode ser implementada, apesar de, na prática, isso não ser necessário. A função `CopiaString()` apresentada a seguir é funcionalmente equivalente à função **strcpy()** da biblioteca padrão de C.

```
char *CopiaString(char *destino, const char *origem)
{
      /* Guarda endereço do array apontado por 'destino' */
   char *inicioStrDestino = destino;

      /* Copia cada caractere do string 'origem' para o array 'destino' */
   while (*destino++ = *origem++) {
      ; /* Não há mais nada a fazer */
   }

   return inicioStrDestino;
}
```

Se você entendeu o funcionamento da função **ComprimentoStr3()** apresentada na seção anterior, certamente, não terá dificuldades em entender a função **CopiaString()** exibida acima. Nessa última função, novamente, as expressões ***destino++** e ***origem++** referem-se, respectivamente, aos conteúdos das posições de memória para onde apontam os ponteiros **destino** e **origem**. Portanto a expressão:

```
*destino++ = *origem++
```

copia o caractere apontado pelo ponteiro **origem** no conteúdo apontado por **destino** e incrementa os dois ponteiros. O laço **while** termina quando for copiado o caractere nulo; i.e., quando o resultado da aplicação do operador de atribuição for zero. Se a função **CopiaString()** for compilada utilizando GCC com a opção **-Wall** ou outro compilador com uma opção equivalente, será apresentada uma mensagem de advertência alertando o programador para o fato de ele poder ter equivocadamente trocado o operador de igualdade (representado por ==) pelo operador de atribuição (representado por =), que é um engano que ocorre com frequência. Aqui, todavia, não houve nenhum engano e o programador pode ignorar seguramente essa mensagem de advertência.

O uso de **const** na declaração do segundo parâmetro da função **strcpy()** garante que o conteúdo do string usado como doador de caracteres na operação de cópia não é modificado pela função.

Não custa nada salientar novamente que, quando a função **strcpy()** é chamada, o primeiro parâmetro deve ser o endereço de um array com capacidade suficiente para conter o resultado da operação, e não um ponteiro qualquer para o tipo **char**. Essa é uma causa comum de erros de programação entre programadores inexperientes em C.

9.5.5 Concatenação de Strings: strcat()

A função **strcat()** acrescenta os caracteres de um string (segundo parâmetro) ao final de outro string armazenado num array de caracteres (primeiro parâmetro). Essa função retorna o endereço do array recebido como primeiro parâmetro e seu protótipo é:

```
char *strcat(char *destino, const char *origem)
```

Um exemplo de uso de **strcat()** é apresentado no seguinte programa:

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

int main(void)
{
   char  str1[] = "Boa ";
   char *str2 = "bola";
   char  resultado[20];
   strcat(strcpy(resultado, str1), str2);
   printf("\nResultado da concatenacao: \"%s\"\n", resultado);

   return 0;
}
```

Note, nesse programa, que o primeiro parâmetro da chamada de **strcat()** é o valor retornado pela função **strcpy()**, que é o endereço do array `resultado[]`. Conforme foi afirmado na Seção 9.5.4, na maioria das vezes, não é necessário usar o valor retornado pela função **strcpy()**, mas, nesse caso, se ele não fosse usado seriam necessárias duas linhas de instrução para obter a concatenação desejada:

```
strcpy(resultado, str1);
strcat(resultado, str2);
```

O retorno da função **strcat()** é semelhante ao da função **strcpy()** e seu uso é recomendado quando o resultado da concatenação é necessário como parte de outra operação, como ocorreu com o valor retornado pela função **strcpy()** no programa acima.

A implementação de uma função que realiza concatenação, como **strcat()**, é simples, desde que você tenha entendido a implementação da função `CopiaString()` da Seção 9.5.4. A função `ConcatenaStrings()`, apresentada a seguir, é funcionalmente equivalente a **strcat()**.

```
char *ConcatenaStrings(char *destino, const char *origem)
{
      /* Guarda endereço do array apontado por 'destino' */
   char *inicioDestino = destino;

      /* Faz 'destino' apontar para o primeiro caractere adiante de '\0' */
   while (*destino++)
      ; /* VAZIO */

      /* Na saída do último laço while, 'destino' aponta para um  */
      /* caractere adiante de '\0' e é preciso fazê-lo retroceder */
   destino--;

      /* Copia cada caractere do string origem no array destino */
   while (*destino++ = *origem++)
      ; /* Não há mais nada a fazer */

   return inicioDestino;
}
```

Essa última função apresenta duas instruções a mais do que a função `CopiaString()` vista na Seção 9.5.4. Essas instruções têm como objetivo fazer o ponteiro `destino` apontar para o caractere terminal do string contido no primeiro parâmetro da função. As instruções mencionadas são:

```
while (*destino++)
   ; /* VAZIO */
```

e

```
destino--;
```

O laço **while** acima encerra quando a expressão `*destino++` resulta em zero. Mas, como o operador sufixo de incremente está sendo usado, quando essa expressão resulta em zero, o ponteiro `destino` apontará para um caractere adiante do caractere `'\0'`. Por isso, a instrução `destino--` se faz necessária. O restante do corpo da função `ConcatenaStrings()` é semelhante ao corpo da função `CopiaString()`.

Novamente, lembre-se que o array cujo endereço é passado como primeiro parâmetro de **strcat()** deve ter espaço suficiente para conter o resultado da concatenação, pois, caso contrário, haverá corrupção de memória (v. Seção 8.9.3).

9.5.6 Comparação de Strings: strcmp() e strcoll()

Em programação, dois strings são comparados da seguinte maneira:

- Os caracteres correspondentes dos dois strings são comparados um a um. Isto é, o primeiro caractere do primeiro string é comparado com o primeiro caractere do segundo string, o segundo caractere do primeiro string é comparado com o segundo caractere do segundo string e assim por diante.
- Se, durante a comparação, não for encontrada nenhuma diferença entre caracteres e os dois strings contêm o mesmo número de caracteres, eles serão considerados iguais. Isto é, dois strings são iguais quando eles têm o mesmo comprimento e contêm exatamente os mesmos caracteres nas respectivas posições.
- Se, durante a comparação, forem encontrados dois respectivos caracteres diferentes, o string que contém o *menor* caractere é considerado menor e o outro string será considerado maior (v. abaixo). Isto é, um string é menor do que (i.e., **precede**) outro se, quando eles são comparados caractere a caractere, encontra-se um caractere no primeiro string que precede o caractere correspondente no segundo string. Por outro lado, um string é maior do que (i.e., **sucede**) outro se ele não é nem igual nem menor do que o outro.

Ordem de colação é um conjunto de normas que ditam como caracteres são ordenados. Se os caracteres em questão consistirem apenas de letras, ordem de colação é exatamente o mesmo que **ordem alfabética**. Em programação, a ordem de colação mais comum (e menos útil para ordenar strings) é aquela obtida comparando-se caracteres de acordo com os valores inteiros atribuídos a eles no código de caracteres utilizado. Essa é a **ordem de colação padrão** utilizada na linguagem C, mas ela pode ser alterada (v. adiante).

Na maioria dos códigos de caracteres conhecidos, dígitos precedem letras maiúsculas que, por sua vez, precedem letras minúsculas. As letras maiúsculas e minúsculas sem acentuações são ordenadas de acordo com a ordem alfabética usual. A **Tabela 9–3** apresenta exemplos de strings ordenados usando-se esse tipo de colação de caracteres.

Exemplo	Justificativa
`"copo"` precede `"corpo"`	Os primeiros caracteres em que os strings diferem são `'p'` e `'r'` e `'p'` precede `'r'`.
`"Carol"` precede `"Carolina"`	O primeiro string termina antes do segundo.
`"Maria"` precede `"maria"`	Os strings diferem no primeiro caractere e `'M'` precede `'m'`.
`"Zebra"` precede `"abelha"`	Os strings diferem no primeiro caractere e `'Z'` (maiúsculo) precede `'a'` (minúsculo).
`"José"` precede `"João"`	Os primeiros caracteres em que os strings diferem são `'s'` e `'ã'`. Nos códigos de caracteres que contêm letras acentuadas, os valores inteiros atribuídos a essas letras são maiores do que aqueles atribuídos a letras não acentuadas.

Tabela 9–3: Strings Ordenados Usando Colação de Código de Caracteres

Os dois últimos exemplos na **Tabela 9–3** mostram que ordenação de strings usando valores atribuídos a caracteres num determinado código de caracteres não é muito útil para ordenar strings de modo a satisfazer a expectativa habitual. Entretanto, esse método ainda é útil quando o objetivo é comparar strings para determinar se eles são exatamente iguais ou diferentes.

Função strcmp()

A função **strcmp()** é utilizada para comparar strings usando a ordem padrão de colação de caracteres e tem como protótipo:

```
int strcmp(const char *str1, const char *str2)
```

Os parâmetros de **strcmp()** são ambos strings e essa função retorna o seguinte:

- Zero, se os strings são iguais
- Um valor negativo, se o primeiro string for menor do que o segundo
- Um valor positivo, se o primeiro string for maior do que o segundo

O seguinte programa apresenta vários exemplos de uso da função **strcmp()**:

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

/****
 * ClassificaStrings(): Compara strings usando strcmp() e exibe o resultado na tela
 *
 * Parâmetros: s1, s2 (entrada) - strings que serão comparados
 *
 * Retorno: Nada
 ****/
void ClassificaStrings(const char *s1, const char *s2)
{
   int   compara;

      /* Obtém o resultado da comparação de s1 e s2 */
   compara = strcmp(s1, s2);

      /* Apresenta o resultado da comparação */
   if (!compara) {
      printf("\n\"%s\" e \"%s\" sao iguais", s1, s2);
   } else {
      printf("\n\"%s\" %s \"%s\"", s1, compara < 0 ? "precede" : "sucede", s2);
   }
}
int main(void)
{
      /* Chama ClassificaStrings() para apresentar  */
      /* resultados de comparação de vários strings */
   ClassificaStrings("copo", "corpo");
   ClassificaStrings("Carol", "Carolina");
   ClassificaStrings("Maria", "maria");
   ClassificaStrings("Zebra", "abelha");
   ClassificaStrings("José", "João");
   ClassificaStrings("Bola", "Bola");
   ClassificaStrings("Maria", "Carol");

   return 0;
}
```

O resultado desse programa quando ele é executado é o seguinte:

```
"copo" precede "corpo"
"Carol" precede "Carolina"
"Maria" precede "maria"
"Zebra" precede "abelha"
"José" precede "João"
"Bola" e "Bola" sao iguais
"Maria" sucede "Carol"
```

No último programa, a função `ClassificaString()` é responsável pela classificação dos strings comparados. Essa função chama **strcmp()** para comparar os strings recebidos como parâmetros e atribui o resultado à variável `compara`. Então, essa variável é usada para classificar os strings usando a seguinte instrução **if**:

```
if (!compara) {
   printf("\n\"%s\" e \"%s\" sao iguais", s1, s2);
} else {
   printf( "\n\"%s\" %s \"%s\"", s1,
           compara < 0 ? "precede" : "sucede", s2 );
}
```

O interessante aspecto a ser observado nessa instrução é o uso do operador condicional formando uma expressão que é um dos parâmetros da segunda chamada de **printf**():

```
compara < 0 ? "precede" : "sucede"
```

De acordo com essa expressão, quando o valor da variável `compara` é menor do que zero, o parâmetro (string) a ser escrito é `"precede"`; caso contrário, será escrito `"sucede"`. Sem o uso do operador condicional, seria necessário aninhar um segundo **if** na parte **else**. Ou seja, seria necessário escrever a última instrução **if** como:

```
if (!compara) {
   printf("\n\"%s\" e \"%s\" sao iguais", s1, s2);
} else if (compara < 0) {
   printf( "\n\"%s\" precede \"%s\"", s1, s2);
} else {
   printf( "\n\"%s\" sucede \"%s\"", s1, s2);
}
```

Quando se comparam strings contendo apenas letras não acentuadas e não se deseja levar em consideração diferenças entre letras maiúsculas e respectivas letras minúsculas, podem-se comparar os strings convertidos em maiúsculas ou minúsculas e, assim, obter um resultado mais satisfatório, como mostra o programa da Seção 9.10.9.

Localidade e a Função strcoll()

Conforme foi discutido, a função **strcmp**() não é conveniente para ordenar strings numa linguagem contendo caracteres acentuados, como ocorre com português. Por exemplo, como foi mostrado, de acordo com a comparação efetuada por **strcmp**(), `"José"` precede `"João"`, o que seria um disparate em qualquer lista alfabética em língua portuguesa. Acontece, porém, que, num código de caracteres, todos os caracteres estão associados a valores inteiros distintos entre si e, quando compara caracteres, a função **strcmp**() utiliza exatamente esses valores. Além disso, quando o código de caracteres utilizado inclui caracteres acentuados, esses caracteres possuem valores inteiros associados maiores do que caracteres não acentuados. Por exemplo, `'z'` precede `'ã'`, pois o primeiro caractere não possui acento, enquanto o segundo caractere é acentuado.

Em programação, uma **localidade** especifica um conjunto de convenções para apresentação de dados. Por exemplo, localidades diferentes podem apresentar informações sobre datas e valores monetários de formas diferentes. Esse tópico tem complexidade que está bem além do escopo deste livro e será abordado superficialmente aqui; i.e., o leitor será provido apenas com o conhecimento necessário para ser capaz de usar corretamente a função **strcoll**(), que é semelhante a **strcmp**(), mas leva em consideração critérios de colação específicos de uma determinada localidade. Ou seja, tanto os parâmetros dessas funções quanto os valores retornados por elas têm as mesmas interpretações, mas esses valores podem ser diferentes dependendo da localidade utilizada.

Na **localidade padrão**, que qualquer programa escrito em C usa se não houver indicação em contrário (v. abaixo), as funções **strcoll**() e **strcmp**() funcionam exatamente do mesmo modo. Por exemplo, se você substituir as chamadas de **strcmp**() por chamadas de **strcoll**() nos exemplos apresentados acima, obterá os mesmos resultados. Portanto, para obter o efeito desejado com o uso da função **strcoll**(), é necessário, antes de chamá-la, alterar a localidade padrão.

A escolha mais simples e natural de localidade é aquela correntemente utilizada pelo sistema operacional sob a supervisão do qual o programa é executado. Para efetuar essa alteração, é preciso, em primeiro lugar, incluir o cabeçalho da biblioteca padrão que lida com localidades:

```
#include <locale.h>
```

Em seguida, altera-se a localidade por meio de uma chamada da função **setlocale()**, como mostrado a seguir:

```
setlocale(LC_COLLATE, "");
```

A função **setlocale()** possui dois parâmetros. O primeiro deles é representado por uma constante simbólica que informa qual é a categoria de localidade (v. adiante) que se deseja alterar e o outro é um string que informa o nome da nova localidade. Quando o segundo parâmetro é um string vazio, como na chamada de **setlocale()** acima, entende-se que ele faz referência à localidade ora sendo utilizada pelo sistema operacional em vigor.

As informações presentes numa localidade são agrupadas em **categorias**, cada uma das quais concentrada num aspecto de formatação de dados. Isso permite que formatos de apresentação de dados que fazem parte de categorias diferentes sejam alterados independentemente. Por exemplo, pode-se alterar o formato de apresentação de datas e horas independentemente do formato de apresentação de valores monetários. Em C, categorias são representadas por constantes simbólicas e cada uma delas começa com **LC_**.

Aqui, a categoria de localidade que interessa é aquela que define a ordem de colação de caracteres e essa categoria é representada pela constante simbólica **LC_COLLATE** definida em `<locale.h>`. Como se deseja usar a localidade corrente do sistema operacional, passa-se o string vazio, representado por um par de aspas, como segundo parâmetro na chamada de **setlocale()**. Essa chamada só precisa ser efetuada uma vez [normalmente, no corpo da função **main()**] e antes da primeira chamada de **strcoll()**, como mostra o programa a seguir:

```
#include <stdio.h>
#include <string.h>
#include <locale.h>

/****
 * ClassificaStrings2(): Compara strings usando strcoll() e exibe o resultado na tela
 *
 * Parâmetros: s1, s2 (entrada) - strings que serão comparados
 *
 * Retorno: Nada
 ****/
void ClassificaStrings2(const char *s1, const char *s2)
{
   int   compara;

      /* Obtém o resultado da comparação dos strings s1 e s2 usando strcoll() */
   compara = strcoll(s1, s2);

      /* Apresenta o resultado da comparação */
   if (!compara) {
      printf("\n\"%s\" e \"%s\" sao iguais", s1, s2);
   } else {
      printf("\n\"%s\" %s \"%s\"", s1, compara < 0 ? "precede" : "sucede", s2);
   }
}

int main(void)
{
      /* Usa a localidade do sistema operacional */
   setlocale(LC_COLLATE, "");
```

```
      /* Chama ClassificaStrings2() para apresentar */
      /* resultados de comparação de vários strings */
   ClassificaStrings2("copo", "corpo");
   ClassificaStrings2("Carol", "Carolina");
   ClassificaStrings2("Maria", "maria");
   ClassificaStrings2("Zebra", "abelha");
   ClassificaStrings2("José", "João");
   ClassificaStrings2("Bola", "Bola");
   ClassificaStrings2("Maria", "Carol");

   return 0;
}
```

Esse programa apresenta o seguinte resultado:

```
"copo" precede "corpo"
"Carol" precede "Carolina"
"Maria" sucede "maria"
"Zebra" sucede "abelha"
"José" sucede "João"
"Bola" e "Bola" sao iguais
"Maria" sucede "Carol"
```

Compare o resultado apresentado pelo último programa, que usa **strcoll()** para comparar strings, com o resultado do penúltimo programa, que usa de **strcmp()** com a mesma finalidade. Observe que os últimos resultados exibidos satisfazem a expectativa de quem domina a língua portuguesa. Contudo, se a instrução:

```
setlocale(LC_COLLATE, "");
```

que altera a localidade do último programa, for removida, os resultados dos dois programas serão idênticos.

9.5.7 Casamento de Strings: strstr()

A função **strstr()** recebe dois strings como parâmetros de entrada e retorna o endereço da primeira ocorrência do segundo string no primeiro. Se nenhum **casamento** for encontrado, essa função retorna **NULL**. O protótipo de **strstr()** é:

```
char *strstr(const char *string1, const char *string2)
```

Como exemplo de uso de **strstr()**, considere o seguinte programa:

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

int main(void)
{
   char *ptr1 = "Bom Dia Brasil";
   char *ptr2 = "dia";
   char *ptr3 = "Dia";
   char *posicao;

   posicao = strstr(ptr1, ptr2);

   if (!posicao) {
      printf( "\n\"%s\" nao foi encontrado em \"%s\"\n", ptr2, ptr1 );
   } else {
      printf( "\nConteudo de \"%s\" a partir de \"%s\":"
              " \"%s\"\n", ptr1, ptr2, posicao );
   }
```

```
   posicao = strstr(ptr1, ptr3);

   if (!posicao) {
      printf( "\n\"%s\" nao foi encontrado em \"%s\"\n",
              ptr3, ptr1 );
   } else {
      printf( "\nConteudo de \"%s\" a partir de \"%s\":"
              " \"%s\"\n", ptr1, ptr3, posicao );
   }
   return 0;
}
```

Quando executado, o último programa apresenta o seguinte resultado na tela:

```
"dia" nao foi encontrado em "Bom Dia Brasil"

Conteudo de "Bom Dia Brasil" a partir de "Dia": "Dia Brasil"
```

9.5.8 Procurando um Caractere num String: strchr() e strrchr()

Função strchr()

A função **strchr()** procura a *primeira* ocorrência de um caractere num string e seu protótipo é:

```
char *strchr(const char *string, int caractere)
```

Quando bem sucedida, essa função retorna o endereço da primeira ocorrência do caractere (segundo parâmetro) no string (primeiro parâmetro). Se o caractere não for encontrado, essa função retorna **NULL**. Como exemplo de uso dessa função considere o seguinte programa:

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

int main(void)
{
   char *str1 = "Isso e' um teste";
   char *str2 = "Bola";
   char *posicao;
   int   c = 't';

   posicao = strchr(str1, c);

   if (!posicao) {
      printf( "\nO caractere \'%c\' nao foi encontrado em \"%s\"\n", c, str1 );
   } else {
      printf( "\nConteudo de \"%s\" a partir do primeiro "
              "\ncaractere \'%c\': \"%s\"\n", str1, c, posicao );
   }

   posicao = strchr(str2, c);

   if (!posicao) {
      printf("\nO caractere \'%c\' nao foi encontrado em \"%s\"\n", c, str2);
   } else {
      printf( "\nConteudo de \"%s\" a partir do primeiro "
              "caractere \'%c\': \"%s\"\n", str2, c, posicao );
   }

   return 0;
}
```

Quanto executado, esse programa escreve na tela:

```
Conteudo de "Isso e' um teste" a partir do primeiro
caractere 't': "teste"

O caractere 't' nao foi encontrado em "Bola"
```

É interessante notar que a função **strchr()** pode ser usada para obter o endereço do caractere terminal de um string `str` como: `strchr(str, '\0')`. Levando esse fato em consideração, pode-se escrever uma função que exibe na tela um string invertido, como mostra o seguinte programa:

```
#include <stdio.h>
#include <string.h>
void ExibeStringInvertido(const char *str)
{
   char *ptr;

      /* Faz 'ptr' apontar para o caractere terminal do string */
   ptr = strchr(str, '\0');

   putchar('\"'); /* Escreve as aspas iniciais do string */

      /* Escreve cada caractere a partir do caractere */
      /* que antecede '\0' até o primeiro caractere   */
   while (ptr != str) {
         /* Escreve o caractere anterior àquele  */
         /* para onde 'ptr' aponta correntemente */
      putchar(*--ptr);
   }

   putchar('\"'); /* Escreve as aspas finais do string */
}
int main(void)
{
   char *str = "Roma";

   printf( "\nString original: \"%s\"", str);
   printf( "\nString invertido: ");
   ExibeStringInvertido(str);

   return 0;
}
```

O último programa contém comentários que espera-se possam dirimir quaisquer dúvidas. Quando executado, esse programa exibe o seguinte na tela:

```
String original: "Roma"
String invertido: "amoR"
```

Função strrchr()

A função **strrchr()** é semelhante à função **strchr()**, mas, ao contrário dessa última função, **strrchr()** procura a *última* ocorrência de um caractere (segundo parâmetro) num string (primeiro parâmetro). O protótipo da função **strrchr()** é:

```
char *strrchr(const char *string, int caractere)
```

Quando encontra o caractere procurado, essa função retorna seu endereço. Se o caractere não for encontrado, essa função retorna **NULL**. O seguinte programa apresenta um exemplo de uso da função **strrchr()**:

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

int main(void)
{
   char *str1 = "Isso e' um teste";
   char *str2 = "Bola";
   char *posicao;
   int   c = 't';

   posicao = strrchr(str1, c);

   if (!posicao) {
      printf( "\nO caractere \'%c\' nao foi encontrado em \"%s\"\n", c, str1 );
   } else {
      printf( "\nConteudo de \"%s\" a partir do ultimo "
              "caractere \'%c\': \"%s\"\n", str1, c, posicao );
   }

   posicao = strrchr(str2, c);

   if (!posicao) {
      printf("\nO caractere \'%c\' nao foi encontrado em \"%s\"\n", c, str2);
   } else {
      printf( "\nConteudo de \"%s\" a partir do ultimo "
              "caractere \'%c\': \"%s\"\n", str2, c, posicao );
   }

   return 0;
}
```

Quanto executado, esse último programa escreve na tela:

```
Conteudo de "Isso e' um teste" a partir do ultimo caractere 't': "te"
O caractere 't' nao foi encontrado em "Bola"
```

Assim como **strchr()**, a função **strrchr()** também leva em consideração o caractere terminal de string, de modo que a chamada `strrchr(str, '\0')` tem o mesmo efeito que a chamada `strchr(str, '\0')`. Usando-se esse conhecimento, pode-se calcular o comprimento de um string sem precisar chamar a função **strlen()** (v. Seção 9.5.3) como mostra o programa a seguir:

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

int main(void)
{
   char  ar[20] = "Bola";
   char *posicaoTerminal;

   posicaoTerminal = strchr(ar, '\0');

   printf( "\nTamanho do string \"%s\" usando strchr():"
           " %d\n", ar, posicaoTerminal - ar );

   posicaoTerminal = strrchr(ar, '\0');

   printf( "Tamanho do string \"%s\" usando strrchr():"
           " %d\n", ar, posicaoTerminal - ar );

   return 0;
}
```

Quando executado, esse programa produz o seguinte na tela:

```
Tamanho do string "Bola" usando strchr(): 4
Tamanho do string "Bola" usando strrchr(): 4
```

9.5.9 Separando um String em Partes (Tokens): strtok()

Um **token** é uma sequência de caracteres considerada como uma unidade que possui significado próprio num determinado contexto. Tokens num string são identificados por caracteres **separadores** que os delimitam. Por exemplo, um comando de um sistema operacional pode ser considerado como um string composto de tokens, como o comando `ls` de sistemas da família Unix a seguir:

```
ls -alt
```

Nesse exemplo, o comando é composto por dois tokens: (1) `ls`, que é o nome do comando, e (2) `-alt`, que são as opções do comando. Nesse caso, o separador de tokens é espaço em branco. O comando completo é considerado um string, mas cada token que o compõe também é considerado um string (ou, melhor, um substring).

A função **strtok()** divide um string em tokens e seu protótipo é:

```
char *strtok(char *str, const char *separadores)
```

Nesse protótipo, os parâmetros têm os seguintes significados:

- `str` é o string a ser dividido em partes (tokens). **NB**: Esse string não pode ser constante; i.e., um programa contendo uma chamada de **strtok()** tendo como primeiro parâmetro um string constante poderá ser abortado.
- `separadores` é um string contendo os caracteres que separam as partes

A primeira chamada de **strtok()** retorna o endereço do primeiro token encontrado no string `str` e um caractere terminal de string é colocado nesse parâmetro ao final do referido token. Chamadas subsequentes dessa função usando **NULL** como primeiro parâmetro retornarão os tokens seguintes até que nenhuma deles seja remanescente no string original. Quando nenhum token é encontrado, a função **strtok()** retorna **NULL**. Por exemplo, supondo que o string a ser dividido em tokens seja `"Um Dois Tres"` e que o único separador de tokens seja `' '` (espaço em branco), a **Figura 9–1** ilustra três chamada consecutivas da função **strtok()**; na primeira delas, o string `"Um Dois Tres"`, armazenado num array, é passado como primeiro parâmetro da função, enquanto, nas demais chamadas, o primeiro parâmetro é **NULL**. A **Figura 9–1** mostra, para cada chamada da função **strtok()**, o endereço retornado e o conteúdo do string original alterado por essa função.

FIGURA 9–1: FUNCIONAMENTO DA FUNÇÃO strtok()

Considere o seguinte programa como exemplo de uso da função **strtok()**:

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

int main(void)
{
   char string[] = "Este e' um string com 7 tokens.";
   char separadores[] = " ."; /* Note o espaço em branco */
   char *token;
   int   i = 0;

   printf("\nO string a ser separado em tokens e': "
          "\n\t\"%s\"\n", string);
   printf("\nOs tokens sao:\n\n");

   token = strtok(string, separadores);

   while (token) {
      printf("\tToken %d: \"%s\"\n", ++i, token);
      token = strtok(NULL, separadores);
   }

   printf("\nString original alterado por strtok(): "
          "\n\t\"%s\"\n", string);

   return 0;
}
```

Quando executado, esse programa escreve o seguinte na tela:

```
O string a ser separado em tokens e':
        "Este e' um string com 7 tokens."

Os tokens sao:

        Token 1: "Este"
        Token 2: "e'"
        Token 3: "um"
        Token 4: "string"
        Token 5: "com"
        Token 6: "7"
        Token 7: "tokens"

String original alterado por strtok():
        "Este"
```

Deve-se chamar atenção para o fato de a função **strtok()** modificar o string passado como primeiro parâmetro, conforme mostra a **Figura 9–1**. Portanto, se for necessário preservar o string original, faça uma cópia dele antes de chamar essa função. Deve-se notar ainda que o string contendo os separadores (i.e., o segundo parâmetro da função) pode ser diferente em duas chamadas sucessivas, embora isso não seja comum.

9.6 A Função main()

A função **main()**, que tem sido intensamente utilizada até aqui, é uma função com certas características especiais. A presença dessa função num programa em C é obrigatória em programas hospedados; i.e., programas que são executados sob intermediação de um sistema operacional (v. **Seção 3.17**). Essa função é sempre a primeira função a ser executada no programa e, quando ela retorna, o programa é encerrado.

Conforme já foi visto informalmente em diversos exemplos apresentados ao longo deste livro, a função **main()** possui como protótipo:

```
int main(void)
```

Agora, outra característica importante da função **main()** é que ela pode receber dois parâmetros do sistema operacional no qual o programa é executado. Esses parâmetros estão associados a strings que acompanham a invocação do programa via console e são denominados **argumentos de linha de comando**[2]. O protótipo da função **main()** que incorpora esses dois parâmetros é o seguinte:

```
int main(int argc, char *argv[])
```

O primeiro parâmetro recebido pela função **main()** quando o programa que a contém é executado é tradicionalmente denominado *argc* e representa o número de argumentos presentes na linha de comando do sistema operacional quando esse programa é invocado. O segundo parâmetro fornecido pelo sistema operacional, tradicionalmente denominado *argv*, consiste num array de strings que armazena os argumentos presentes na linha de comando quando o programa é invocado[3].

Os argumentos passados para um programa incluem seu nome e cada argumento (string) que constitui o comando de execução do programa deve ser separado de outro por meio de um ou mais espaços em branco:

nome-do-programa $argumento_1$ $argumento_2$ ... $argumento_n$

Na verdade, o modo como argumentos são passados para um programa depende do sistema operacional utilizado e não é especificado pelo padrão de C. O formato apresentado aqui é o mais comumente utilizado. Sistemas operacionais das famílias Windows/DOS e Unix usam esse formato.

Quando a execução de um programa é iniciada, seu nome é armazenado como primeiro elemento no array `argv[]` e os demais strings presentes na linha de comando serão armazenados consecutivamente nesse array. Ao parâmetro `argc` será automaticamente atribuído o número de elementos do array `argv[]`. Por exemplo, se houver três argumentos na linha de comando, além do nome do programa, `argv[]` terá quatro elementos e `argc` assumirá `4` como valor.

Como exemplo concreto de uso de argumentos de linha de comando, considere o seguinte programa:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
   int  i;

   printf("argc  = %d\n", argc);

   for(i = 0;  i < argc; i++) {
      printf("argv[%d] = %s\n", i, argv[i]);
   }

   return 0;
}
```

Quando executado, esse programa apresenta na tela o valor de `argc` e os strings armazenados em `argv[]`. Por exemplo, suponha que o nome desse programa após ser convertido em executável é `exemplo`. Então, como resultado da execução desse programa por meio do comando:

```
exemplo azul preto branco
```

[2] Argumentos de linha de comando também são denominados *parâmetros de linha de comando*, mas, para evitar que o termo *parâmetro*, nesse contexto, seja confundido com o mesmo termo usado no contexto de funções, aqui, será dada preferência a *argumento*.

[3] O nome *argc* é derivado de *argument count* em inglês ou, equivalentemente em português, *número de argumentos*. Por outro lado, *argv* é derivado de *argument vector* em inglês, que é equivalente em português a *vetor de argumentos*.

seria produzido como saída:

```
argc = 4
argv[0] = exemplo
argv[1] = azul
argv[2] = preto
argv[3] = branco
```

É importante salientar que, nos dois protótipos da função **main()** aceitos pelo padrão ISO de C, o tipo de retorno é **int**. Isso quer dizer que, apesar de alguns compiladores permitirem que **main()** seja definida com tipo de retorno **void**, usar este tipo de retorno não é portável.

9.7 Classificação e Transformação de Caracteres

9.7.1 Classificação de Caracteres: Funções isX()

No cabeçalho `<ctype.h>`, são declaradas funções que classificam caracteres de acordo com diversas categorias, tais como letras, dígitos e espaços em branco. Todas essas **funções de classificação de caracteres** têm nomes começando com *is* e recebem um parâmetro do tipo **int** representando o caractere que será classificado. Cada uma delas verifica se o caractere recebido como parâmetro satisfaz uma determinada propriedade e retorna um valor diferente de zero, se esse for o caso ou zero, se o caractere não satisfaz a propriedade a que se refere a função. As funções de classificação de caracteres mais comumente utilizadas são brevemente descritas na Tabela 9–4.

Função	Retorna um valor diferente de zero se o parâmetro representar...
isalnum()	*um caractere alfanumérico (i.e., dígito ou letra)*
isalpha()	*uma letra*
isblank()	`' '` *ou* `'\t'`
isdigit()	*um dígito*
islower()	*uma letra minúscula*
ispunct()	*um símbolo de pontuação*
isspace()	*um espaço em branco qualquer, incluindo quebra de linha (*`'\n'`*)*
isupper()	*uma letra maiúscula*

Tabela 9–4: Funções de Classificação de Caracteres mais Comuns

Para utilizar qualquer das funções descritas na Tabela 9–4, deve-se incluir o cabeçalho `<ctype.h>`.

A função `EhStringAlfabetico()` no programa a seguir usa a função **isalpha()** para verificar se um string contém apenas letras.

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

/****
 * EhStringAlfabetico(): Checa se um string contém apenas letras
 *
 * Parâmetros: str (entrada) - o string a ser checado
 *
 * Retorno: 1, se o string for alfabético; 0, em caso contrário
 ****/
int EhStringAlfabetico(const char *str)
{
```

```
      /* O laço while é encerrado quando um caractere  */
      /* que não é letra é encontrado (inclusive '\0') */
   while (isalpha(*str++))
      ; /* Vazio */

   --str; /* Faz 'str' apontar para o caractere que causou o encerramento do laço */

      /* Se todos os caracteres são alfabéticos, 'str' está apontando para o */
      /* caractere '\0'. Nesse caso, !*str resulta em 1. Caso contrário,     */
      /* 'str' está apontando para um caractere que não é alfabético e o     */
      /* resultado de !*str é 0.                                             */
   return !*str;
}

int main(void)
{
   char *s1 = "bola";
   char *s2 = "bola?";

   printf( "O string \"%s\" %s alfabetico\n", s1,
           EhStringAlfabetico(s1) ? "e'" : "nao e'" );

   printf( "O string \"%s\" %s alfabetico\n", s2,
           EhStringAlfabetico(s2) ? "e'" : "nao e'" );

   return 0;
}
```

Quando executado, esse programa exibe o seguinte resultado:

```
O string "bola" e' alfabetico
O string "bola?" nao e' alfabetico
```

É importante salientar que o funcionamento de qualquer função do módulo ctype é dependente de localidade (v. Seção 9.5.6). Por exemplo, o caractere `'ã'` não é considerado letra na localidade padrão de acordo com a função **isalpha()**, mas passará a ser considerado como letra se você alterar a localidade padrão para uma localidade de língua portuguesa, como demonstram os seguintes exemplos.

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

int main(void)
{
   if (isalpha('ã')) {
      printf("\nEsse caractere e' letra\n");
   } else {
      printf("\nEsse caractere NAO e' letra\n");
   }
   return 0;
}
```

Quando esse programa é executado, o resultado que ele apresenta é:

```
Esse caractere NAO e' letra
```

Agora, considere o seguinte exemplo semelhante:

```
#include <stdio.h>
#include <ctype.h>
#include <locale.h>

int main(void)
```

```
{
   setlocale(LC_CTYPE, ""); /* Usa a localidade corrente do sistema operacional */

   if (isalpha('ã')) {
      printf("\nEsse caractere e' letra\n");
   } else {
      printf("\nEsse caractere NAO e' letra\n");
   }

   return 0;
}
```

Apesar de ser parecido com o programa anterior, esse último programa apresenta como resultado:

```
Esse caractere e' letra
```

O último programa apresenta a resposta esperada e incorpora as seguintes alterações com relação ao programa anterior:

- Ele inclui o cabeçalho `<locale.h>` (v. Seção 9.5.6).
- Ele efetua a seguinte chamada da função **setlocale()**:

  ```
  setlocale(LC_CTYPE, "");
  ```

  Essa chamada de **setlocale()** indica que o programa deve usar a categoria de localidade **LC_CTYPE** da localidade vigente no sistema operacional (supostamente de língua portuguesa — v. Seção 9.5.6). Essa categoria de localidade é associada a classificação de caracteres.

Outros exemplos de funções de classificação de caracteres usadas com frequência serão apresentadas adiante (veja, p. ex., as Seção 9.10.1 e Seção 9.10.2).

9.7.2 Transformação de Caracteres: tolower() e toupper()

Além das funções de classificação de caracteres apresentadas na Seção 9.7.1, o módulo ctype também provê duas funções de **transformação de caracteres** (assim denominadas pelo padrão ISO de C). A função **tolower()** retorna a letra minúscula correspondente a um dado caractere e a função **toupper()** retorna a letra maiúscula correspondente a um dado caractere. Se o único parâmetro que cada uma dessas funções recebe não for letra, o retorno é o próprio caractere recebido como parâmetro. Lembre-se que, como as demais funções do módulo ctype, essas duas funções dependem de localidade. Por exemplo, você não conseguirá transformar em letra maiúscula o caractere `'ã'` na localidade padrão (v. Seção 9.5.6).

O seguinte programa utiliza a função **toupper()** para converter um string em letras maiúsculas:

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

/****
 *
 * ConverteEmMaiusculas(): Converte as letras de um string em maiúsculas.
 *                         Caracteres que não são letras não são afetados.
 *
 * Parâmetros: str (entrada/saída): o string
 *
 * Retorno: O endereço do string recebido como parâmetro
 *
 ****/
char* ConverteEmMaiusculas(char *str)
{
```

```
   char *p = str; /* Guarda início do string */

      /* Converte em maiúscula cada caractere que é letra */
   while (*str) {
      *str = toupper(*str); /* Converte caractere atual */
      ++str; /* Passa para o próximo caractere */
   }
   return p; /* Retorna o endereço do string */
}

int main(void)
{
   char  s1[] = "Zebra";
   char  *s2 = "anta";

      /* A instrução a seguir causa um erro de lógica */
   printf( "\nString \"%s\" convertido em maiusculas:"
           " \"%s\"\n", s1, ConverteEmMaiusculas(s1) );

      /* A instrução a seguir causa um erro de execução */
   printf( "\nString \"%s\" convertido em maiusculas:"
           " \"%s\"\n", s2, ConverteEmMaiusculas(s2) );

   return 0;
}
```

A função `ConverteEmMaiusculas()` desse programa usa o seguinte laço **while** para converter eventuais letras minúsculas do string recebido como parâmetro em letras maiúsculas:

```
while (*str) {
   *str = toupper(*str);
   ++str;
}
```

Esse laço acessa cada caractere do string recebido como parâmetro e converte-o em letra maiúscula (se ele for letra minúscula, obviamente). A conversão em maiúsculas ocorre na linha:

```
*str = toupper(*str);
```

9.8 Ordem de Avaliação de Parâmetros

A função `ConverteEmMaiusculas()` apresentada como último exemplo da Seção 9.7.2 é perfeitamente correta, mas as duas chamadas dessa função efetuadas na função **main()** estão incorretas. Para confirmar essa afirmação, compile e execute o programa da Seção 9.7.2 e você poderá obter o seguinte como resultado:

```
String "ZEBRA" convertido em maiusculas: "ZEBRA"
[Programa abortado]
```

A segunda chamada da função `ConverteEmMaiusculas()` (v. Seção 9.7.2) parece ser obviamente incorreta, visto que ela causa o aborto do programa, porém o erro na primeira chamada pode ter passado despercebido.

O fato é que, quando o programador escreveu a instrução:

```
printf( "\nString \"%s\" convertido em maiusculas:"
        " \"%s\"\n", s1, ConverteEmMaiusculas(s1) );
```

muito provavelmente, ele esperava que a saída correspondente do programa fosse:

```
String "Zebra" convertido em maiusculas: "ZEBRA"
```

Ou seja, o programador esperava que o string `"Zebra"` (antes da conversão) fosse escrito antes do string `"ZEBRA"` (depois da conversão). O problema aqui é que chamada de função é representada por um operador (v. **Seção 10.4**) que, como quase todos os operadores de C, não possui ordem de avaliação definida (v. **Seção 3.6.2**). Isso significa que a ordem com que os parâmetros são avaliados numa chamada de função não é especificada. No caso específico do exemplo acima, o último parâmetro foi avaliado antes do penúltimo e, por isso, quando o string `s1` (penúltimo parâmetro) foi escrito, ele já havia sido convertido em maiúsculas. Para corrigir o primeiro problema apresentado pelo programa, é necessário dividir a chamada da função **printf()** mencionada em duas, como, por exemplo:

```
printf( "\nString \"%s\"", s1 );
printf( "convertido em maiusculas: \"%s\"\n", ConverteEmMaiusculas(s1) );
```

O conselho que se deve seguir para evitar erros como o primeiro erro associado ao programa acima é:

Recomendação ***Nunca faça suposições sobre a ordem de avaliação de parâmetro numa chamada de função.***

O segundo erro que ocorre no programa da **Seção 9.7.2** já foi discutido na **Seção 9.3**, mas aqui ele aparece sob disfarce. Isto é, na **Seção 9.3** foi recomendado que strings constantes devem realmente ser considerados constantes e não devem ter seus conteúdos alterados, sob o risco de causarem aborto de programa. No exemplo em questão, o string `"anta"`, diferentemente de `"Zebra"`, é um string constante e, quando ele é passado como parâmetro para a função `ConverteEmMaiusculas()`, ela tenta alterar seu conteúdo, o que causa o aborto do programa em consequência de **violação de memória**.

Violação de memória é um conceito semelhante a corrupção de memória (v. **Seção 8.9.3**), mas, rigorosamente, esses conceitos são diferentes. Violação de memória sempre causa aborto de programa, ora em virtude de tentativa de acesso a uma região de memória proibida (p. ex., o espaço de endereçamento de outro programa), ora em consequência de tentativa de operação que não é permitida no espaço de endereçamento do próprio programa em que ela ocorre (esse é o caso do programa em questão).

É importante observar que o parâmetro da função `ConverteEmMaiusculas()` é declarado como **char** * e a ausência de **const** nessa declaração sugere que a função pode alterar o conteúdo apontado por seu parâmetro[4]. E isso realmente ocorre na referida função, conforme foi visto acima. Portanto, para evitar problemas como esse, siga sempre a seguinte recomendação:

Recomendação ***Se uma função não usa const na declaração de um parâmetro formal que representa um string, não é seguro ligar esse parâmetro a um string constante.***

Para evitar esquecimento, permita que o compilador lhe advirta com relação a essa recomendação, declarando ponteiros para string constantes com o uso de **const**, como, por exemplo:

```
const char *s2 = "anta";
```

Declarando o ponteiro `s2` dessa maneira, o compilador apresentará uma mensagem de advertência se houver uma tentativa de usá-lo como um parâmetro para uma função que não se compromete a tratar o conteúdo apontado por `s2` como constante.

[4] A ausência de **const** na declaração de um parâmetro representado por um ponteiro pode também indicar negligência por parte de um programador que não segue boas normas de estilo. Mas, esse não é o caso aqui.

9.9 Conversão de Strings Numéricos em Números

9.9.1 Função atoi()

A função **atoi()** faz parte do módulo stdlib da biblioteca padrão de C e é usada para converter um string num valor do tipo **int**, desde que o string permita tal conversão. O protótipo dessa função é:

```
int atoi(const char *str)
```

Nesse protótipo, `str` é o string que se deseja converter em inteiro e o retorno é o valor do tipo **int** resultante da conversão do string. Se o string não puder ser convertido, o retorno é zero e, quando isso ocorre, a interpretação do valor retornado é ambígua, pois esse valor pode indicar que houve erro de conversão ou que o resultado da conversão foi realmente zero.

Observações sobre a função atoi():

- O string recebido como parâmetro pode conter um sinal.
- A conversão prossegue até que o primeiro caractere que não possa fazer parte de um número inteiro seja encontrado.
- Se ocorrer overflow (v. Seção 4.11.7), o resultado será indefinido.

O seguinte programa demonstra o uso da função **atoi()**.

```
#include <stdio.h>   /* printf() */
#include <stdlib.h>  /* atoi()   */

int main(void)
{
   char *str = "1234";

   printf( "\n>>> String \"%s\" convertido em int: %d\n", str, atoi(str) );

   return 0;
}
```

O resultado apresentado pelo programa acima é:

```
>>> String "1234" convertido em int: 1234
```

9.9.2 Função strtod()

A função **strtod()**, declarada em `<stdlib.h>`, converte strings em números reais do tipo **double** e seu protótipo é:

```
double strtod(const char *str, char **final)
```

Nesse protótipo, os parâmetros têm os seguintes significados:

- `str` — string a ser convertido.
- `final` — deve ser **NULL** ou o endereço de um ponteiro que, ao final da conversão, apontará para o primeiro caractere do string que não foi convertido. Se todos os caracteres do string forem usados na conversão, esse ponteiro apontará para o caractere terminal do string (v. exemplo adiante).

A função **strtod()** retorna o valor do tipo **double** resultante da conversão, se essa conversão for possível ou zero, se nenhuma conversão for possível.

Um string válido como primeiro parâmetro de **strtod()** pode incluir:

- Espaços em branco em seu início
- Sinal (i.e., `+` ou `-`)

- Ponto decimal
- `e` ou `E` (indicando o uso de notação científica — v. Seção 3.5.2)

Se nenhuma conversão for possível, a função **strtod()** retorna `0.0`, que obviamente, é um valor válido do tipo **double**. Assim, para verificar se ocorreu erro de conversão, deve-se checar o parâmetro `final`. Isto é, se, ao retorno da função, esse parâmetro estiver apontando para o início do string original, pode-se concluir que não houve nenhuma conversão. Por outro lado, se, ao término da conversão, o parâmetro `final` apontar para o caractere terminal do string, todos os caracteres do string foram usados na conversão.

O exemplo a seguir ilustra o uso da função **strtod()**:

```
#include <stdio.h>   /* printf() */
#include <stdlib.h>  /* strtod() */

/****
 * ApresentaConversaoEmDouble(): Converte um string em double usando strtod() e
 *                               exibe o resultado na tela
 *
 * Parâmetros: str (entrada) - string que será convertido
 *
 * Retorno: Nada
 ****/
void ApresentaConversaoEmDouble(const char *str)
{
   double      d;
   char       *p; /* Apontará para o caractere no qual a conversão parou */

      /* Tenta efetuar a conversão do string em double */
   d = strtod(str, &p);

      /* Verifica se foi possível alguma conversão e apresenta o resultado */
   if (p != str) { /* Houve conversão */
      printf( "\n\t>>> Conversao do string: \"%s\"" em double: %4.3G\n", str, d );

         /* Verifica se restaram caracteres a ser convertidos */
      if (*p) { /* Restaram caracteres */
         printf( "\n\t    Faltou converter os caracteres: \"%s\"\n", p );
      } else { /* Não restaram caracteres */
         printf("\n\t   Todos os caracteres foram convertidos\n");
      }
   } else { /* Não houve conversão */
      printf( "\n\t>>> Conversao do string:\n\t    \"%s\""
              "\n\t    em double nao foi possivel\n", str );
   }
}

int main(void)
{
   const char *str1 = "1.6E-19",
              *str2 = "2.54cm equivalem a 1 polegada",
              *str3 = "Carga do eletron: 1.6E-19";

   ApresentaConversaoEmDouble(str1);
   ApresentaConversaoEmDouble(str2);
   ApresentaConversaoEmDouble(str3);

   return 0;
}
```

Esse programa produz o seguinte resultado na tela:

```
>>> Conversao do string: "1.6E-19" em double: 1.6E-019
   Todos os caracteres foram convertidos
>>> Conversao do string: "2.54cm equivalem a 1 polegada" em double: 2.54
    Faltou converter os caracteres: "cm equivalem a 1 polegada"
>>> Conversao do string: "Carga do eletron: 1.6E-19"
    em double nao foi possivel
```

9.10 Exemplos de Programação

9.10.1 Leitura de Nomes

Problema: (a) Escreva uma função que lê nomes; i.e., strings contendo apenas letras e espaços em branco. (b) Escreva um programa que teste a função solicitada no item (a).

Solução de (a):

```
/****
 * LeNome(): Lê um string via teclado e garante que ele só
 *           contém letras e espaços em branco
 *
 * Parâmetros: nome (saída) - array que conterá o string lido
 *             tam (entrada) - número de elementos do array
 *
 * Retorno: Número de caracteres digitados além do limite permitido
 *
 * Observações:
 *      1. Caracteres digitados além do limite especificado serão ignorados
 *      2. O único espaço em branco permitido num nome é ' '
 *      3. Esta função não testa se o string digitado é
 *         constituído apenas por espaços em branco
 ****/
int LeNome(char *nome, int tam)
{
   int   remanescentes; /* Número de caracteres digitados além do permitido */
   char *p; /* Usado para testar a validade do string */

   while (1) {
      printf("\nDigite o nome (max = %d letras):\n\t> ", tam-1);
      remanescentes = LeString(nome, tam);

         /* Verifica se o string lido é vazio */
      if (!*nome) {
         printf("\a\n\t>> Um nome nao pode ser vazio <<\n");
         continue; /* Salta o restante do laço */
      }

      /*** Verifica se o string lido só contém letras e espaços em branco ***/

         /* O ponteiro p será usado para testar o string */
      p = nome;

         /* O laço encerra quando p apontar para o caractere terminal do string ou */
         /* quando for encontrado um caractere que nem é letra nem espaço em branco */
      while (*p) {
            /* Se for encontrado um caractere que não é letra */
            /* ou espaço em branco encerra este laço          */
```

```
         if( !isalpha(*p) && *p != ' ' ) {
            break;
         }

         p++; /* Aponta para o próximo caractere */
      }

         /* Se p apontar para o caractere terminal do */
         /* string, este foi aprovado no teste acima  */
      if (!*p) {
         break; /* String só contém letras e espaços */
      } else { /* String foi reprovado no teste */
         printf("\a\n\t>>> Um nome deve conter apenas <<<"
                "\n\t>>> letras e espacos em branco <<<\n");
      }
   }

   return remanescentes;
}
```

Análise:

- A função `LeNome()` usa o ponteiro local `p` para testar o string introduzido pelo usuário, pois se o próprio parâmetro `nome` fosse utilizado com esse intuito, ele não estaria mais apontando para o início do array que armazena o resultado caso uma nova tentativa de leitura fosse necessária.
- Essa função não testa se o string lido é constituído apenas por espaços em branco, de forma que o usuário pode digitar um nome que será invisível quando exibido. Contudo, corrigir esse problema é relativamente fácil e é deixado como exercício para o leitor.

Solução de (b):

```
/****
 * main(): Testa função LeNome()
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
{
   char umNome[MAX_NOME + 1];
   int  resto; /* Caracteres excedentes na leitura */

      /* Apresenta o programa */
   printf( "\n\t>>> Este programa le nomes contendo apenas "
           "letras e espacos em branco\n" );

      /* Lê o nome e atribui a 'resto' o número de caracteres excedentes na leitura */
   resto = LeNome(umNome, MAX_NOME + 1);
      /* Apresenta o nome que o programa efetivamente leu */
   printf("\n\t>>> O nome aceito foi: %s\n", umNome);

      /* Verifica se o usuário digitou caracteres demais */
   if (resto) {
      printf("\n\t>>> %d caracteres foram desprezados\n", resto);
   }

   return 0;
}
```

Para completar o programa, as seguintes linhas devem ser incluída em seu início:

```
#include <stdio.h>   /* printf()   */
#include <ctype.h>   /* isalpha()  */
#include "leitura.h" /* LeString() */

#define MAX_NOME 20 /* Número máximo de caracteres num nome */
```

Exemplo de execução do programa:

```
        >>> Este programa le nomes contendo apenas letras e espacos em branco

Digite o nome (max = 20 letras):
        > Micro$oft

        >>> Um nome deve conter apenas <<<
        >>> letras e espacos em branco <<<

Digite o nome (max = 20 letras):
        >

        >> Um nome nao pode ser vazio <<

Digite o nome (max = 20 letras):
        > Oftalmotorrinolaringologista

        >>> O nome aceito foi: Oftalmotorrinolaring

        >>> 8 caracteres foram desprezados
```

9.10.2 Leitura de Números de Identificação

Preâmbulo: Um **número de identificação** é um string contendo apenas dígitos e com tamanho prefixado. Brasileiros vivem às voltas com números de identificação: CPF, cédula de identidade, PIS/PASEP, título de eleitor e sabem-se lá quantos outros. Apesar de receberem essa denominação popular, rigorosamente falando, números de identificação não são realmente *números*. Isto é, em jargão de programação, eles são mais apropriadamente denominados **strings numéricos**.

Problema: (a) Escreva uma função que lê um número de identificação. (b) Escreva um programa que lê uma matrícula de quatro dígitos usando a função solicitada no item (a).

Solução de (a):

```
/****
 *
 * LeIdentidade(): Lê um número de identificação via teclado
 *
 * Parâmetros:
 *     id[] (saída) - array que conterá o número lido
 *     tamArray (entrada) - número de elementos do array id[]
 *     prompt (entrada) - string que representa a porção inicial
 *                        do prompt a ser apresentado
 *
 * Retorno: O endereço do array apontado por id
 *
 * Observação: Assume-se que o tamanho do número de identificação
 *             é igual ao tamanho do array id[] que a armazenará menos um
 *
 ****/
char *LeIdentidade(char *id, int tamArray, const char *prompt)
{
   int   resto; /* Número de eventuais caracteres excedentes */
   char *p; /* Usado para testar o string */
```

```
      /* O laço encerra quando for lido um string válido */
   while (1) {
         /* Apresenta prompt */
      printf("\n%s com exatamente %d digitos:\n\t> ", prompt, tamArray - 1);

      resto = LeString(id, tamArray); /* Lê entrada do usuário */

         /* Checa se o usuário digitou caracteres em excesso */
      if (resto) {
         printf("\a\n\t>>> Excesso de caracteres <<<\n");
         continue; /* Faz nova tentativa */
      }

         /* Verifica se foram digitados menos caracteres do que esperado. Note o */
         /* uso do operador de conversão (int) para converter o valor retornado  */
         /* por strlen(), que é do tipo size_t, e, assim, evitar comparação de   */
         /* um valor sem sinal e outro com sinal.                                */
      if ( (int)strlen(id) != tamArray - 1 ) {
         printf("\a\n\t>>> Numero incompleto <<<\n");
         continue; /* Faz nova tentativa */
      }

         /************************************************/
         /* Neste ponto, o tamanho do número está correto. */
         /* Resta testar se ele só contém dígitos.         */
         /************************************************/

      p = id; /* O ponteiro p será usado para testar o string */
         /* O laço encerra quando p apontar para o caractere terminal do  */
         /* string ou quando for encontrado um caractere que não é dígito */
      while (*p) {
            /* Se for encontrado um caractere que não é dígito encerra este laço */
         if(!isdigit(*p)) {
            break; /* Encerra o laço interno */
         }

         p++; /* Aponta para o próximo caractere */
      }

         /* Se p apontar para o caractere terminal do */
         /* string, este foi aprovado no teste acima  */
      if (!*p) {
         break; /* Número de identificação válido */
      } else { /* Número de identificação inválido */
         printf("\a\n\t>>> \'%c\' nao e' digito <<<\n", *p);
      }
   }

   return id;
}
```

Análise:

- O operador de conversão explícita **(int)** foi aplicado sobre o valor retornado pela função **strlen()** para evitar uma comparação entre um valor inteiro *sem* sinal [i.e., aquele retornado por **strlen()**] e um valor inteiro *com* sinal. Erros decorrentes dessa mistura maligna estão entre os mais difíceis de encontrar num programa (v. **Seção 8.5**).
- A função `LeIdentidade()` usa um ponteiro auxiliar (variável local `p`) para verificar a validade do string introduzido pelo usuário, pois, se o próprio parâmetro `id` fosse utilizado com esse intuito, ele

não estaria mais apontando para o início do array que armazena o resultado, caso uma nova tentativa de leitura fosse necessária, nem ao final da função.

Solução de (b):

```
/****
 *
 * main(): Lê uma matrícula de quatro dígitos
 *
 * Parâmetros:
 *
 * Retorno: Zero
 *
 ****/
int main(void)
{
   char matr[TAM_MATR + 1]; /* Array que armazenará a matrícula */

      /* Apresenta o programa */
   printf( "\n\t>>> Este programa le uma matricula com 4 digitos.\n" );

      /* Lê um número de matrícula desprezando */
      /* o retorno da função LeIdentidade()    */
   LeIdentidade(matr, TAM_MATR + 1, "Digite uma matricula");

      /* Apresenta a matrícula lida */
   printf("\n\t>>> Matricula digitada: %s\n", matr);

   return 0;
}
```

Para completar o programa, deve-se acrescentar o seguinte ao seu início:

```
#include <stdio.h>   /* printf()   */
#include <string.h>  /* strlen()   */
#include <ctype.h>   /* isdigit()  */
#include "leitura.h" /* LeString() */

#define TAM_MATR  4 /* Número de caracteres numa matrícula */
```

Exemplo de execução do programa:

```
        >>> Este programa le uma matricula com 4 digitos

Digite uma matricula com exatamente 4 digitos:
        > 12

        >>> Numero incompleto <<<

Digite uma matricula com exatamente 4 digitos:
        > 12ab

        >>> 'a' nao e' digito <<<

Digite uma matricula com exatamente 4 digitos:
        > 4444

        >>> Matricula digitada: 4444
```

9.10.3 Checando a Validade de PIS/PASEP

Preâmbulo: Um **número de PIS/PASEP** é um string constituído apenas de dígitos e suas partes constituintes são exibida na **Figura 9–2**.

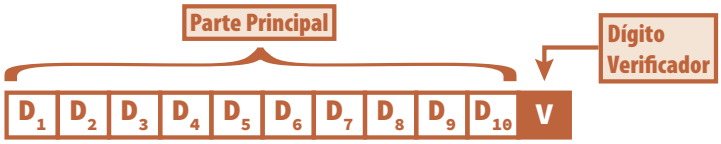

FIGURA 9–2: NÚMERO DE PIS/PASEP

Para verificar se um número de PIS/PASEP é válido, segue-se o seguinte procedimento:

1. Os inteiros representados pelos dígitos do número principal são multiplicados, respectivamente, pelos seguintes valores (**pesos**): `3`, `2`, `9`, `8`, `7`, `6`, `5`, `4`, `3` e `2`.
2. Os valores obtidos no passo anterior são somados.
3. Calcula-se o resto da divisão da soma obtida no passo anterior por `11`.
4. Para um número de PIS/PASEP ser considerado válido, o valor do dígito verificador deve ser igual a `11` menos o resto da divisão obtido no passo anterior.

Problema: Escreva um programa que lê um string via teclado e verifica se ele constitui um número de PIS/PASEP válido.

Solução:

```
#include <stdio.h>   /* printf()   */
#include <string.h>  /* strlen()   */
#include <ctype.h>   /* isdigit()  */
#include "leitura.h" /* LeString() */

   /* Número de dígitos num número de PIS/PASEP */
#define TAMANHO_PIS 11

/****
 * main(): Lê um string e verifica se ele constitui um número de PIS/PASEP válido
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
{
   char pis[TAMANHO_PIS + 1];
   int  i, soma = 0,
        pesos[] = {3, 2, 9, 8, 7, 6, 5, 4, 3, 2};

      /* Apresenta o programa */
   printf( "\n\t>>> Este programa verifica se um numero"
           "\n\t>>> de PIS/PASEP esta' correto.\n" );

      /* Lê o número de PIS/PASEP. A função LeIdentidade() garante */
      /* que o string é do tamanho correto e só contém dígitos.    */
   LeIdentidade(pis, TAMANHO_PIS + 1, "Digite o PIS/PASEP");

      /* Agora, calcula-se a soma ponderada dos dígitos */
   for (i = 0; i < TAMANHO_PIS - 1; ++i) {
      soma = soma + pesos[i]*(pis[i] - '0');
   }

      /* Neste ponto, i é igual a TAMANHO_PIS - 1, que é o índice do último */
      /* dígito (i.e., o dígito verificador). Então, checa-se se o valor do */
      /* dígito verificador é igual a 11 menos o resto da divisão da soma   */
      /* ponderada obtida por 11.                                           */
```

```
   if (pis[i] - '0' == TAMANHO_PIS - soma%TAMANHO_PIS) {
      printf("\n\t>>> O numero de PIS e' valido\n");
   } else {
      printf("\n\t>>> O numero de PIS e' INVALIDO\n");
   }
   return 0;
}
```

Análise: A função **main**() chama a função `LeIdentidade()`, que foi discutida na Seção 9.10.2, e, por isso, sua definição foi omitida da listagem apresentada.

Exemplo de execução do programa:

```
        >>> Este programa verifica se um numero de PIS/PASEP esta' correto.

Digite o PIS/PASEP com exatamente 11 digitos:
        > 18434219765

        >>> O numero de PIS e' valido
```

Observação: Esse programa foi testado e aprovado com números de PIS/PASEP reais, mas o número usado nesse exemplo de execução foi gerado por outro programa especificamente criado com esse propósito. Portanto qualquer coincidência com um número de PIS/PASEP realmente existente terá sido mera coincidência (como nos filmes).

9.10.4 Centralizando Strings na Tela

Problema: (a) Escreva uma função que apresenta strings centralizados na tela. (b) Escreva um programa que testa a função solicitada no item (a).

Solução de (a):

```
/****
 * CentralizaString(): Apresenta um string centralizado na tela
 *
 * Parâmetros:
 *     string (entrada) - string a ser centralizado
 *     largura (entrada) - largura da linha na qual o string será centralizado
 *
 * Retorno: Nada
 ****/
void CentralizaString(const char *string, int largura)
{
   int tamanhoStr, /* Tamanho do string */
       coluna, /* Conta espaços em branco que devem preceder o string na tela */
       espacosEsquerda; /* Número de espaços que devem preceder */
                        /* a escrita  do string na tela         */

   tamanhoStr = strlen(string); /* Calcula o tamanho do string */

      /* Calcula o número de espaços que devem ser deixados em branco */
   espacosEsquerda = (largura - tamanhoStr)/2;

      /* Preenche os espaços em branco que devem preceder o string */
   for(coluna = 0; coluna < espacosEsquerda; coluna++) {
      putchar(' ');
   }

      /* Escreve string após espaços em branco iniciais */
   printf("%s\n", string);
}
```

Solução de (b): A solução para esse item é relativamente fácil e é deixada como exercício para o leitor. Você poderá conferir sua solução no site do livro (*www.ulysseso.com/ip*).

9.10.5 Criptografia Chinfrim

Preâmbulo: **Criptografia** consiste num conjunto de técnicas utilizadas para cifrar arquivos de modo a evitar que pessoas não autorizadas tenham acesso aos conteúdos desses arquivos. A necessidade de segurança cada vez maior em virtude do crescente fluxo de informações em redes de computadores tem estimulado o surgimento de algoritmos de criptografia cada vez mais sofisticados. Um dos métodos mais simples de criptografia consiste na utilização de um **código de substituição direta**. Essa técnica rudimentar funciona apenas para arquivos de texto e consiste em substituir cada caractere (letra) no texto original por um caractere (único) correspondente que faz parte da **chave criptográfica**, que é um string. Por exemplo, a letra `'A'` seria substituída pelo primeiro caractere (letra) no string que representa a chave criptográfica, a letra `'B'` seria substituída pelo segundo caractere nesse string e assim por diante. Para essa técnica funcionar, é essencial que haja uma relação biunívoca entre os caracteres no texto original e os caracteres na chave de criptografia (caso contrário, como você traduziria de volta ao original um documento criptografado?).

Problema: Escreva uma função denominada `Criptografa()` que criptografa um string passado como primeiro parâmetro utilizando uma chave passada como segundo parâmetro e retorna o string criptografado. Assuma que apenas as letras de `'a'` a `'z'` (minúsculas) são mapeadas. (b) Escreva uma função denominada `Decifra()` que decifra um string passado como primeiro parâmetro utilizando uma chave passada como segundo parâmetro e retorna o string decifrado. Assuma que apenas as letras de `'a'` a `'z'` (minúsculas) foram mapeadas no string criptografado. (c) Escreva uma função **main()** que recebe um string como entrada e oferece opções para criptografá-lo ou decifrá-lo usando a chave: `"tfhxqjemupidckvbaolrzwgnsy"`. O programa deve encerrar quando o usuário digitar `[ENTER]` como string. **NB**: O fato de a chave fazer parte do próprio programa é apenas uma simplificação do problema. Um programa mais realístico solicitaria uma chave para criptografar ou decifrar um documento.

Solução de (a):

```
/****
 * Criptografa(): Criptografa um string usando outro string como chave
 * Parâmetros:
 *      str (entrada/saída): string que será criptografado
 *      chave (entrada): a chave usada na criptografia
 *
 * Retorno: O endereço do string criptografado
 *
 * Observação: Apenas letras minúsculas são criptografadas
 ****/
char *Criptografa(char *str, char const *chave)
{
   char *ptr = str; /* Guarda o endereço do string */

   /*************************************************************************/
   /* Substitui cada caractere no string que é letra minúscula pelo caractere */
   /* que se encontra na chave na posição dada pela diferença entre o valor   */
   /* inteiro do caractere e o valor de 'a'.                                  */
   /*************************************************************************/

   while (*str) {
      if (islower(*str)) { /* Apenas letras minúsculas são criptografadas */
```

```
         /* No array que contém a chave, o caractere que substituirá 'a'       */
         /* encontra-se no índice 0, o substituto de 'b' encontra-se no índice */
         /* 1, e assim por diante. Em geral, o substituto de um caractere c    */
         /* encontra-se no índice dado por c - 'a'. Assim, o substituto do     */
         /* caractere apontado por 'str' no array chave[] encontra-se no       */
         /* índice dado por: *str - 'a'.                                       */
         /*                                                 */

            /* Substitui o caractere apontado por 'str' pelo */
            /* caractere correspondente no array chave[]     */
         *str = chave[*str - 'a'];
      }

      str++; /* Passa para o próximo caractere */
   }

   return ptr; /* O endereço do string foi guardado em ptr */
}
```

Análise: Leia os comentários distribuídos na função, pois eles devem ser suficientes para seu entendimento.

Solução de (b):

```
/****
 * Decifra(): Decifra um string usando outro string como chave
 *
 * Parâmetros:
 *      str (entrada/saída) - string que será decifrado
 *      chave (entrada) - a chave criptográfica
 *
 * Retorno: O endereço do string decifrado
 *
 * Observação: Apenas letras minúsculas são decifradas
 ****/
char *Decifra(char *str, char const *chave)
{
   char *ptr = str; /* Guarda o endereço do string */

   /*********************************************************/
   /* Encontra o índice no array chave[] de cada caractere que é letra    */
   /* no string criptografado,  soma o valor do caractere que se encontra */
   /* nesse índice ao valor do caractere 'a' e substitui o caractere      */
   /* correspondente no string criptografado pelo resultado obtido.       */
   /*********************************************************/

   while (*str) {
      if (islower(*str)) { /* Apenas letras minúsculas foram criptografadas */
         /* Encontra o índice do caractere apontado por 'str'  */
         /* no array chave[]usando a expressão:                */
         /*         strchr(chave, *str) - chave                */
         /* Somando-se esse índice a 'a', obtém-se o caractere */
         /* que substituirá aquele apontado por 'str'.         */

         *str = strchr(chave, *str) - chave + 'a';
      }

      str++; /* Passa para o próximo caractere */
   }

   return ptr; /* O endereço do string foi guardado em ptr */
}
```

Análise: Leia os comentários distribuídos na função, pois eles devem ser suficientes para seu entendimento.

Solução de (c):

```
/****
 *
 * main(): Lê frases via teclado e criptografa-as ou decifra-as
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 *
 ****/
int main(void)
{
   const char *chave = "tfhxqjemupidckvbaolrzwgnsy";
   char        str[TAMANHO_ARRAY];
   int         op;

      /* Apresenta o programa */
   printf( "\n\t>>> Este programa criptografa e decifra frases"
           "\n\t>>> escritas com letras minusculas (apenas)." );

      /* O laço encerra quando o usuário digitar [ENTER] */
   while (1) {
      printf("\n\nIntroduza uma frase ou apenas [ENTER] para encerrar:\n\t> ");
      LeString(str, TAMANHO_ARRAY);

         /* Verifica se o string é vazio */
      if (!*str) {
         break; /* Usuário digitou apenas [ENTER] */
      }

         /* Lê a opção do usuário */
      printf( "\nEscolha C para criptografar ou D para decifrar o texto:\n\t> " );
      op = LeOpcao("cCdD");

         /* Criptografa ou decifra, de acordo com a opção do usuário */
      if (op == 'c' || op == 'C') {
         printf( "\nTexto criptografado:\n\t> %s", Criptografa(str, chave) );
      } else {
         printf( "\nTexto decifrado:\n\t> %s", Decifra(str, chave) );
      } /* if */
   } /* while */

   printf( "\n>>> A chave criptografica usada foi:\n\t> %s."
           "\n>>> Por favor, memorize-a.\n", chave );

   return 0;
}
```

Complemento do programa:

```
#include <stdio.h>   /* printf()               */
#include <string.h>  /* strchr()               */
#include <ctype.h>   /* islower()              */
#include "leitura.h" /* LeString() e LeOpcao() */

   /* Tamanho do array que armazena os strings introduzidos pelo usuário */
#define TAMANHO_ARRAY 50
```

Exemplo de execução do programa:

```
        >>> Este programa criptografa e decifra frases
        >>> escritas com letras minusculas (apenas).

Introduza uma frase ou apenas [ENTER] para encerrar:
        > O rato roeu a roupa do rei de Roma.

Escolha C para criptografar ou D para decifrar o texto:
        > c

Texto criptografado:
        > O otrv ovqz t ovzbt xv oqu xq Rvct.

Introduza uma frase ou apenas [ENTER] para encerrar:
        > O otrv ovqz t ovzbt xv oqu xq Rvct.

Escolha C para criptografar ou D para decifrar o texto:
        > d

Texto decifrado:
        > O rato roeu a roupa do rei de Roma.

Introduza uma frase ou apenas [ENTER] para encerrar:
        > [ENTER]

>>> A chave criptografica usada foi:
        > tfhxqjemupidckvbaolrzwgnsy.

>>> Por favor, memorize-a.
```

Observação: O exemplo explorado nesta seção tem caráter didático apenas e não representa criptografia séria (por isso, é adjetivada como *chinfrim*). Portanto não utilize o último programa para criptografar nada que requeira segurança de qualquer natureza.

9.10.6 Ocorrências de Letras numa Palavra

Problema: Escreva um programa que conta o número de ocorrências de letras numa palavra. **Observação**: Caracteres acentuados ou cedilha não são aceitos como letras.

Solução:

```
#include <stdio.h>   /* printf()               */
#include <ctype.h>   /* isalpha() e  toupper() */
#include "leitura.h" /* LeituraFacil           */

#define TAM_MAX_PALAVRA  30 /* Tamanho máximo de uma palavra */
#define NUMERO_DE_LETRAS 26 /* Numero de letras no alfabeto */

/****
 * ContaLetrasEmString(): Conta o número de ocorrências de cada letra de um string
 *
 * Parâmetros:
 *      str (entrada) - string cujas letras serão contadas
 *      contaLetras (entrada/saída) - array que contém a contagem de letras
 *
 * Retorno: Nada
 ****/
void ContaLetrasEmString(const char* str, int contaLetras[])
{
   int indice; /* Índice no array de ocorrências que corresponde a uma letra */

      /* Examina cada caractere do string apontado por 'str' e */
      /* conta as ocorrências de cada letra encontrada        */
   while (*str) {
```

```
      if (isalpha(*str)) {
            /* Obtém o índice no array de ocorrências que corresponde à letra */
            /* corrente. Esse índice é obtido convertendo-se a letra em       */
            /* maiúscula e subtraindo-se seu valor de 'A'. Assim, 'A' ('a')   */
            /* tem índice 0, 'B' ('b') tem índice 1 etc.                      */
         indice = toupper(*str) - 'A';

            /* Incrementa o número de ocorrências da letra corrente */
         ++contaLetras[indice];
      }

      ++str; /* Passa para o próximo caractere */
   }
}

/****
 * ApresentaFrequenciaDeLetras(): Apresenta o número de ocorrências de cada letra no array
 *
 * Parâmetros: contaLetras[] (entrada) - array contendo o número de ocorrências
 *                                       de cada letra numa palavra
 *
 * Retorno: Nada.
 *
 * Observação: Letras que não aparece na palavra não são levadas em consideração
 ****/
void ApresentaFrequenciaDeLetras(const int contaLetras[])
{
   int ocorrencias, indice, c;

      /* Escreve cabeçalho da tabela */
   printf( "\nLetra\tOcorrencias"
           "\n=====\t===========\n\n" );

      /* Determina o número de ocorrências de cada letra no array contaLetras[] */
   for (c = 'A'; c <= 'Z'; ++c) {

      /*************************************************************************/
      /* IMPORTANTE: Em qualquer código de caracteres, as letras aparecem em */
      /*             ordem alfabética usual.  Mas, isso não significa que    */
      /*             qualquer caractere entre 'A' e 'Z' ou entre 'a' e 'z' é */
      /*             letra. Por isso, a chamada de isalpha() é necessária.   */
      /*************************************************************************/

         /* Verifica se o caractere é uma letra */
      if (isalpha(c)) { /* Caractere é uma letra */
            /* Obtém o índice no array de ocorrências que    */
            /* corresponde à letra, convertendo-se a letra   */
            /* em maiúscula e subtraindo-se seu valor de 'A' */
         indice = toupper(c) - 'A';

            /* Obtém o número de ocorrências da letra */
         ocorrencias = contaLetras[indice];

            /* Apresenta apenas ocorrências diferentes de zero */
         if (ocorrencias != 0) {
            printf("  %c\t %4d\n", c, ocorrencias);
         }
      }
   }
}
```

```
/****
 * main(): Conta o número de ocorrências de letras numa palavra
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
{
   int  i, contaLetras[NUMERO_DE_LETRAS];
   char palavra[TAM_MAX_PALAVRA + 1];

      /* Apresenta o programa */
   printf( "\n\t>>> Este programa determina o numero de"
           "\n\t>>> ocorrencias de cada letra de uma palavra."
           "\n\t>>> Para encerra-lo digite apenas [ENTER].\n" );

      /* O laço encerra quando o usuário digita apenas [ENTER] */
   while (1) {
         /* Lê a palavra */
      printf( "\n\t>>> Digite a palavra (max = %d "
              "caracteres):\n\t\t> ", TAM_MAX_PALAVRA );
      LeString(palavra, TAM_MAX_PALAVRA + 1);

         /* Verifica se o usuário digitou apenas [ENTER] */
      if (!*palavra) { /* Digitou apenas [ENTER] */
         break; /* Encerra o laço */
      }

         /* Zera o array que armazena as frequências de letras */
      for (i = 0; i < NUMERO_DE_LETRAS; ++i) {
         contaLetras[i] = 0;
      }

         /* Efetua a contagem de frequência de letras */
      ContaLetrasEmString(palavra, contaLetras);

         /* Apresenta o resultado */
      ApresentaFrequenciaDeLetras(contaLetras);
   }
      /* Despede-se do usuário */
   printf( "\n\t>>> Obrigado por usar este programa.\n");

   return 0;
}
```

Análise: Leia atentamente os comentários inseridos no programa para melhor compreendê-lo.

Exemplo de execução do programa:

```
>>> Este programa determina o numero de
>>> ocorrencias de cada letra de uma palavra.
>>> Para encerra-lo digite apenas [ENTER].

>>> Digite a palavra (max = 30 caracteres):
        > Anticonstitucionalissimamente
```

CONTINUA

```
Letra   Ocorrencias
=====   ===========

  A          3
  C          2
  E          2
  I          5
  L          1
  M          2
  N          4
  O          2
  S          3
  T          4
  U          1

        >>> Digite a palavra (max = 30 caracteres):
                > [ENTER]

        >>> Obrigado por usar este programa.
```

9.10.7 Aparando Strings

Problema: (a) Escreva uma função que remove espaços em branco no início de um string. (b) Escreva uma função que remove espaços em branco no final de um string. (c) Escreva uma função que remove espaços em branco tanto no início quanto no final de um string. (d) Escreva um programa para testar as funções solicitadas nos itens anteriores.

Solução de (a):

```
/****
 * RemoveBrancosInicio(): Remove os espaços em branco [de acordo
 *                        com isspace()] no início de um string
 *
 * Parâmetros: str (entrada/saída) - string que terá seus espaços
 *                                   em branco inicias removidos
 *
 * Retorno: Endereço do string modificado (i.e., sem espaços em branco iniciais)
 ****/
char *RemoveBrancosInicio(char *str)
{
   char *p;
   int   i = 0;

      /* Se 'str' for NULL ou o string estiver   */
      /* vazio, esta função não tem nada a fazer */
   if (!str || !*str) {
      return str; /* Nada a fazer */
   }

      /* Faz p apontar para o início do string */
   p = str;

      /* Enquanto p não aponta para o caractere '\0' e o caractere por  */
      /* ele apontado é espaço,  faz p apontar para o próximo caractere */
   while ( p && isspace(*p) ) {
      ++p;
   }

      /* Se, neste ponto, p estiver apontando para o caractere */
      /* terminal do string, ele só continha espaços em branco */
```

```
   if (!*p) { /* O string só continha espaços */
      *str = '\0'; /* O string ficará vazio */
   } else { /* p aponta para o primeiro caractere que não é espaço */

         /* Move cada caractere a partir daquele apontado por p para o início */
      while (*p) {
         str[i] = *p;
         ++i;
         ++p;
      }

         /* O caractere terminal ainda não foi copiado para o string */
      str[i] = '\0';
   }
   return str; /* Retorna o endereço do string */
}
```

Solução de (b):

```
/****
 *
 * RemoveBrancosFim(): Remove os espaços em branco [de acordo
 *                     com isspace()] ao final de um string
 *
 * Parâmetros:
 *     str (entrada/saída) - string que terá seus espaços em branco finais removidos
 *
 * Retorno: Endereço do string sem espaços em branco finais
 *
 ****/
char *RemoveBrancosFim(char *str)
{
   char *p;

      /* Se 'str' for NULL ou o string estiver   */
      /* vazio, esta função não tem nada a fazer */
   if (!str || !*str) {
      return str; /* Nada a fazer */
   }

      /* Faz p apontar para o caractere anterior */
      /* ao caractere terminal do string         */
   p = strchr(str, '\0') - 1;

      /* Enquanto p não aponta para o início do string */
      /* e o caractere por ele apontado é espaço, faz  */
      /* p apontar para o caractere anterior           */
   while ( p != str && isspace(*p) ) {
      --p;
   }

   /*********************************************************/
   /* Se p estiver apontando para o início do string, ele só  */
   /* continha espaços em branco. Nesse caso, o string ficará */
   /* vazio (i.e., contendo apenas o caractere '\0'). Caso    */
   /* contrário, p apontará para o último caractere que não é */
   /* espaço. Assim, coloca-se '\0' no próximo caractere      */
   /* apontado por p (que é um espaço).                       */
   /*********************************************************/
```

```
   if (p == str) { /* String só continha espaços */
      *p = '\0'; /* Torna o string vazio */
   } else { /* p aponta para o último caractere que não é espaço */
      *++p = '\0'; /* Termina o string no próximo caractere */
   }

   return str; /* Retorna o endereço do string */
}
```

Solução de (c):

```
/****
 * RemoveEspacos(): Remove os espaços em branco [de acordo com
 *                  isspace()] no início e no final de um string
 *
 * Parâmetros:
 *     str (entrada/saída) - string que terá seus espaços em
 *                           branco iniciais e finais removidos
 ****/
char *RemoveEspacos(char *str)
{
   str = RemoveBrancosInicio(str); /* Remove espaços iniciais */

      /* Remove espaços finais e retorna o resultado */
   return RemoveBrancosFim(str);
}
```

Solução de (d): A solução para esse item é relativamente fácil e é deixada como exercício para o leitor. Você poderá conferir sua solução no site do livro.

9.10.8 Leitura de Datas com Validação 2

Problema: Escreva um programa que lê uma data no formato `dd/mm/aaaa` ou `dd/mm/aa` e verifica sua validade. Se a data estiver no formato `dd/mm/aa`, o século atual deve ser considerado.

Solução:

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

#include <stdio.h>   /* Entrada e saída             */
#include <string.h>  /* Processamento de strings    */
#include <ctype.h>   /* Classificação de caracteres */
#include "leitura.h" /* LeituraFacil                */

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

#define TAM_ARRAY_DATA          11
#define SECULO_ATUAL          2000
#define PRIMEIRO_ANO_BISSEXTO 1752

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

extern void LeData(int *d, int *m, int *a);
extern int DigitosEmInt( char *digitos, const char *sep,int *num );
extern int ValidaData(char *data, int *d, int *m, int *a);
extern int EhAnoBissexto(int ano);
extern int EhDataValida(int dia, int mes, int ano);

/***************** Definições de Funções *****************/
```

```
/****
 * LeData(): Lê uma data no formato dd/mm/aaaa ou dd/mm/aa
 *
 * Parâmetros: d, m, a (saída) - dia, mes e ano
 *
 * Retorno: Nada
 *
 * Observação: Se o ano só contiver dois dígitos, ele será considerado no século atual
 ****/
void LeData(int *d, int *m, int *a)
{
   char data[TAM_ARRAY_DATA]; /* Armazenará o string contendo a data lida */
   int  resto; /* Caracteres eventualmente remanescentes */
inicio: /* Volta para cá se a data for inválida */
   resto = LeString(data, TAM_ARRAY_DATA);

   while(resto || !ValidaData(data, d, m, a)) {
      printf( "\a\n\t>>> Data invalida. Tente novamente <<<\n\t> " );
      goto inicio; /* Sem espaguete */
   }
}

/****
 * DigitosEmInt(): Extrai um token de um string e converte-o
 *                 em número inteiro (int) positivo
 *
 * Parâmetros:
 *      *digitos (entrada/saída) - string cujo token será extraído e convertido
 *      *sep (entrada) - string contendo os separadores de tokens
 *      *num (saída) - número resultante da conversão
 *
 * Retorno: 0, se a operação for bem sucedida; 1, em caso contrário
 *
 * Observação: Se houver mais de um token a ser processado num mesmo string, a
 *             primeira chamada desta função deve usar o endereço do string
 *             como primeiro parâmetro. Em chamadas subsequentes, esse
 *             parâmetro deve ser NULL.
 ****/
int DigitosEmInt(char *digitos, const char *sep, int *num)
{
   char *token; /* Armazena o token extraído do string */

   token = strtok(digitos, sep); /* Extrai o token */

      /* Se o token não pode ser obtido, o */
      /* número também não pode ser obtido */
   if (!token) {
      return 1; /* Impossível obter número */
   }

      /* Tenta converter o token num inteiro positivo */
   *num = atoi(token);

      /* Se algum caractere deixou de ser convertido, o número é inválido */
   if (*num <= 0) {
      return 1; /* Não é um número inteiro positivo */
   }

   return 0;
}
```

```
/****
 * ValidaData(): Separa uma data contida num string em dia,
 *               mês e ano e tenta validá-la
 * Parâmetros:
 *     data (entrada/saída) - string contendo a data a ser validada
 *     d (saída) - o dia
 *     m (saída) - o mês
 *     a (saída) - o ano
 *
 * Retorno: 1, se a data for válida; 0, se a data não for válida
 *
 * Observação:
 *     1. Se o ano contiver apenas dois dígitos, ele será considerado no século atual
 *     2. O string (primeiro parâmetro) é modificado por esta função.
 ****/
int ValidaData(char *data, int *d, int *m, int *a)
{
      /* Tenta obter o dia */
   if (DigitosEmInt(data, "/", d)) {
      return 0; /* Dia inválido */ /* O dia não pode ser obtido */
   }

      /* Tenta obter o mês */
   if (DigitosEmInt(NULL, "/", m)) {
         /* O mês não pode ser obtido */
      return 0; /* Mês inválido */
   }

      /* Tenta obter o ano */
   if (DigitosEmInt(NULL, "/", a)) {
         /* O ano não pode ser obtido */
      return 0; /* Ano inválido */
   }

      /* Se o ano tem menos de quatro dígitos,  */
      /* assume-se que é um ano do atual século */
   if (*a < 100) {
      *a = *a + SECULO_ATUAL;
   }

      /* A função EhDataValida() completa o serviço */
   return EhDataValida(*d, *m, *a);
}

/****
 * main(): Lê uma data e verifica sua validade
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
{
   int dia, mes, ano;

      /* Apresenta o programa */
   printf( "\n\t>>> Este programa verifica a validade de uma data introduzida no"
           "\n\t>>> formato dd/mm/aaaa ou dd/mm/aa. Se o ano tiver apenas"
           "\n\t>>> dois digitos, sera' \n\t>>> assumido o seculo atual.\n" );
```

```
   printf( "\n\t>>> Digite a data no formato dd/mm/aaaa ou dd/mm/aa:\n\t> " );
   LeData(&dia, &mes, &ano);

   printf( "\n\t>>> A data introduzida foi: \n\t> "
           "%.2d/%.2d/%.2d\n", dia, mes, ano );

   return 0;
}
```

Análise: O programa apresentado utiliza as seguintes funções já vistas:

- **atoi()** — essa função converte um string em número inteiro e foi explorada na Seção 9.9.1.
- `EhAnoBissexto()` e `EhDataValida()` — essas funções foram apresentadas na Seção 7.6.8.

Exemplo de execução do programa:

```
>>> Este programa verifica a validade de uma data introduzida no
>>> formato dd/mm/aaaa ou dd/mm/aa. Se o ano tiver apenas
>>> dois digitos, sera' assumido o seculo atual.

>>> Se o ano tiver apenas dois digitos,
>>> sera' assumido o seculo atual.

>>> Digite a data no formato dd/mm/aaaa ou dd/mm/aa:
> 21/12/10

>>> A data introduzida foi:
> 21/12/2010
```

9.10.9 Comparando Strings sem Diferenciar Maiúsculas e Minúsculas

Problema: (a) Escreva uma função semelhante à função **strcmp()** que compara strings sem levar em consideração diferenças entre letras maiúsculas e minúsculas. (b) Escreva uma função **main()** que testa a função especificada no item (a).

Solução de (a):

```
/****
 * ComparaStr(): Compara dois strings sem levar em consideração
 *               diferenças entre letras maiúsculas e minúsculas
 *
 * Parâmetros: str1 (entrada) - primeiro string a comparar
 *             str2 (entrada) - segundo string a comparar
 *
 * Retorno: = 0, se str1 = str2
 *          < 0, se str1 < str2
 *          > 0, se str1 > str2
 ****/
int ComparaStr(const char *str1, const char *str2)
{
   /*********************************************************************/
   /* Enquanto nenhum dos ponteiros apontar para '\0', compara respectivos */
   /* caracteres convertidos em maiúsculos. Se forem encontrados  dois     */
   /* caracteres diferentes, retorna a diferença entre os valores do       */
   /* primeiro e do segundo caracteres convertidos em maiúsculos.          */
   /*********************************************************************/

      /* O laço encerra quando 'str1' ou 'str2' apontar para */
      /* '\0' ou quando for encontrada uma diferença entre   */
      /* respectivos caracteres convertidos em maiúsculos    */
```

```
   while (*str1 && *str2) {
         /* Se os caracteres maiúsculos sendo comparados diferem, retorna */
         /* a diferença entre os respectivos caracteres maiúsculos        */
      if (toupper(*str1) != toupper(*str2)) {
            /* O uso de toupper() é necessário para produzir a ordem desejada. */
            /* Por exemplo, suponha que os caracteres são 'a' e 'Z'. Então, se */
            /* toupper() não fosse usada, a diferença entre 'a' e 'Z' seria    */
            /* positiva, de modo que o string contendo 'a' seria considerado   */
            /* sucessor daquele que contém 'Z'.                                */
         return toupper(*str1) - toupper(*str2);
      }

         /* Faz cada ponteiro apontar para o próximo caractere a ser comparado */
      ++str1;
      ++str2;
   }

      /*                                                  */
      /* Se 'str1' e 'str2' apontam para seus respectivos caracteres terminais,   */
      /* os strings são iguais (já que eles têm o mesmo tamanho). Caso contrário, */
      /* o string de maior tamanho será o maior.                                  */
      /*                                                  */
   if (!*str1 && !*str2) { /* Strings têm o mesmo tamanho */
      return 0; /* Os strings são considerados iguais */
   }

      /* Neste ponto, sabe-se que um dos ponteiros aponta para '\0', mas o outro   */
      /* não aponta para '\0'. Se o caractere apontado por 'str1' não for'\0',     */
      /* esse ponteiro apontava para o maior string. Caso contrário, ele aponta-   */
      /* va para o menor string. A instrução return a seguir lida com os dois casos */
   return *str1 - *str2;
}
```

Solução de (b): A solução para esse item é relativamente fácil e é deixada como exercício para o leitor. Você poderá conferir sua solução no site do livro.

9.10.10 Criando um Comando de Sistema Operacional

Problema: Escreva um programa denominado, após sua compilação, `somaints` (família Unix) ou `somaints.exe` (Windows/DOS) e que calcula a soma dos valores inteiros introduzidos seguindo o nome do programa na linha de comando do sistema operacional.

Solução:

```
#include <stdio.h>  /* printf()              */
#include <stdlib.h> /* atoi()                */
#include <string.h> /* strchr()              */
#include <ctype.h>  /* isspace() e isdigit() */

/****
 *
 * main(): Calcula a soma dos valores inteiros introduzidos como
 *         argumentos de linha de comando
 *
 * Parâmetros:
 *     argc (entrada) - Número de argumentos de linha de comando
 *     argv (entrada) - Array de strings presentes na linha de
 *                      comando quando o programa o programa é executado
 *
```

```
 * Retorno: Zero, se não ocorrer nenhum erro.
 *          Um valor diferente de zero em caso contrário.
 ****/
int main(int argc, char *argv[])
{
   int   i, valor, soma = 0;

      /* Deve haver pelo menos três argumentos de linha de comando */
   if (argc < 3) {
      printf( "\n\t>>> Este programa deve ser usado assim:"
              "\n\t>>> %s n1 n2 ..., sendo n1, n2, ... "
              "inteiros\n",  argv[0]);
      return 1;
   }

      /* Converte e soma as parcelas representadas */
      /* por argumentos de linha de comando        */
   for (i = 1; i < argc; ++i) {
      valor = atoi(argv[i]);  /* Converte um argumento em inteiro */

      soma = soma + valor; /* Acrescenta o argumento convertido à soma */
   }

   printf("\n\t>>> Soma: %d\n", soma);

   return 0;
}
```

Análise:

- A função **main**() chama **atoi**() para converter os strings recebidos como argumentos em números inteiros. Essa função foi discutida na Seção 9.9.1.
- Note como deve ser a comunicação com o usuário de um programa que recebe argumentos de linha de comando: sucinto, sem apresentação nem despedida.

Exemplo de execução do programa:

```
C:\>somaints -12 32 55ab

        >>> Soma: 75
```

9.10.11 Inserindo um String em Outro

Problema: (a) Escreva uma função que insere um string em outro numa posição especificada. (b) Escreva um programa que, repetidamente, lê dois strings e um número inteiro maior do que um que representa a posição de inserção do segundo string no primeiro string. Então, a função solicitada no item (a) deve ser chamada para efetuar a devida inserção. O programa deve encerrar quando o usuário digitar apenas [ENTER] quando solicitado a introduzir o primeiro string.

Solução de (a):

```
/****
 * InsereString(): Insere um string em outro
 *
 * Parâmetros:
 *     str1 (entrada/saída) - string que receberá a inserção
 *     str2 (entrada) - string a ser inserido
 *     local (entrada) - índice do elemento do string str1 no qual começa a inserção
 *
 * Retorno: Endereço do string resultante
```

```
 *
 * Observação: Se o local especificado para inserção for maior do que o número de
 *             caracteres do string de origem, esta função funciona como strcat()
 ****/
char *InsereString(char *str1, const char *str2, int local)
{
   int tamanho1, /* Tamanho do string que recebe caracteres */
       tamanho2, /* Número de caracteres que serão doados */
       i;

      /* Calcula o tamanho do string no qual será feita a inserção */
   tamanho1 = strlen(str1);

      /* Um string s é indexado de 0 a strlen(s). Portanto, se */
      /* o local da inserção for maior do que ou igual ao seu  */
      /* tamanho deixa-se strcat() completar o serviço.        */
   if (local >= tamanho1) {
      return strcat(str1, str2);
   }

      /* Calcula o número de caracteres que serão inseridos */
   tamanho2 = strlen(str2);

      /* Desloca para adiante cada caractere do string de      */
      /* destino um número de posições igual ao número de      */
      /* caracteres que serão inseridos,  a partir do local de */
      /* inserção. Os últimos caracteres são deslocado antes.  */
   for (i = tamanho1 + 1; i >= local; --i) {
      str1[i + tamanho2] = str1[i];
   }

      /* O espaço para inserção está aberto. Resta     */
      /* copiar os caracteres que devem ser inseridos. */
   for (i = 0; i < tamanho2; ++i) {
      str1[i + local] = str2[i];
   }
   return str1;
}
```

Análise: Os comentários inseridos na própria função `InsereString()` devem ser suficientes para entendê-la.

Solução de (b):

```
/****
 * main(): Insere strings em outros strings nas posições indicadas
 *
 * Parâmetros: Nenhum
 *
 * Retorno: Zero
 ****/
int main(void)
{
   char str1[TAM_ARRAY], str2[TAM_ARRAY];
   int  posicao;

      /* Apresenta o programa */
   printf("\n\t>>> Este programa insere caracteres numa"
          "\n\t>>> cadeia de caracteres numa dada posicao."
          "\n\t>>> Digite [ENTER] como primeira cadeia de"
          "\n\t>>> caracteres para encerrar o programa.\n");
```

```
      /* O laço encerra quando for lido apenas */
      /* [ENTER] como primeiro string          */
   while (1) {
         /* Lê o primeiro string */
      printf("\n\t>>> Digite uma cadeia de caracteres "
             "\n\t    (max = %d digitos): ", TAM_ARRAY - 1);
      (void) LeString(str1, TAM_ARRAY);

         /* Se o string for vazio encerra o laço */
      if (!*str1) { /* Usuário digitou apenas [ENTER] */
         break;
      }

         /* Lê o string a ser inserido */
      printf("\n\t>>> Digite os caracteres a ser inseridos "
             "\n\t    (max = %d digitos): ", TAM_ARRAY - 1);
      (void) LeString(str2, TAM_ARRAY);

         /* O laço encerra quando o usuário digitar um valor inteiro positivo */
      while (1) {
         printf( "\n\t>>> Digite a posicao de insercao: " );
         posicao = LeInteiro();

            /* Usuário comum não indexa strings a partir de   */
            /* zero, como faz a função InsereString(). Assim, */
            /* é necessário decrementar a posição introduzida */
         --posicao;

         if (posicao > 0) {
            break;
         }
         printf("\a\n\t>> O valor deve ser positivo <<\n");
      }

         /* Insere os caracteres de 'str2' em 'str1' na posição especificada */
      InsereString(str1, str2, posicao);

         /* Apresenta o resultado da operação */
      printf( "\n\t>>> Resultado da insercao: \"%s\"\n", str1 );
   }
   printf( "\n\t>>> Obrigado por usar este programa.\n");

   return 0;
}
```

Análise: A função **main()** acima leva em consideração o fato de usuário comum não iniciar contagens a partir de zero, como costumam fazer programadores de C. Logo ela decrementa a posição introduzida pelo usuário antes de chamar a função `InsereString()`.

Para completar o programa, insira o seguinte ao seu início:

```
#include <stdio.h>   /* printf()                */
#include <string.h>  /* strlen() e strcat()     */
#include "leitura.h" /* LeString() e LeInteiro() */

   /* Tamanhos dos arrays que armazenarão os strings */
#define TAM_ARRAY 30
```

Exemplo de execução do programa:

```
>>> Este programa insere caracteres numa
>>> cadeia de caracteres numa dada posicao.
>>> Digite [ENTER] como primeira cadeia de
>>> caracteres para encerrar o programa.

>>> Digite uma cadeia de caracteres
    (max = 29 digitos): bocha

>>> Digite os caracteres a ser inseridos
    (max = 29 digitos): la

>>> Digite a posicao de insercao: 3

>>> Resultado da insercao: "bolacha"

>>> Digite uma cadeia de caracteres
    (max = 29 digitos): [ENTER]

>>> Obrigado por usar este programa.
```

9.11 Exercícios de Revisão

Introdução (Seção 9.1)

1. O que é um string?
2. (a) Todo array de caracteres é um string? (b) Todo string é um array de caracteres?
3. (a) O que é caractere nulo? (b) Qual é o valor inteiro associado ao caractere nulo em qualquer código de caracteres usado em C? (c) Para que serve o caractere nulo?
4. Por que strings são tão importantes em programação?

Armazenamento de Strings em Arrays de Caracteres (Seção 9.2)

5. Se a iniciação do array `ar[]`:
   ```
   char ar[] = {'b', 'o', 'l', 'a', '\0'};
   ```
   é o mesmo que:
   ```
   char  ar[] = "bola";
   ```
   por que existe essa segunda notação para iniciação de arrays de caracteres?
6. A iniciação do array `ar[]` a seguir é legal? Explique.
   ```
   char  ar[3] = "bola";
   ```
7. Na iniciação a seguir, como os elementos do array `ar[]` são iniciados?
   ```
   char  ar[10] = "bola";
   ```
8. (a) A seguinte iniciação do array `ar[]` é legal? (b) Se esse for o caso, qual será o conteúdo do array `ar[]` após essa iniciação?
   ```
   char ar[4] = "bola";
   ```
9. O que será escrito na tela após a execução do trecho de programa abaixo?
   ```
   char  ar[] = "Boa noite",
        *ptr = &ar[1];

   printf("%s", ptr + 4);
   ```
10. (a) Interprete a definição de variável a seguir. (b) Supondo que um ponteiro ocupe 4 bytes, quantos bytes serão alocados em decorrência dessa definição?
    ```
    char *ar[] = {"azul", "vermelho", "branco"};
    ```

Strings Constantes (Seção 9.3)

11. Por que um programa pode ser abortado ao tentar alterar o conteúdo de um string constante?

12. (a) Por que é recomendado o uso de **const** na definição de ponteiros para strings constantes? (b) Dê exemplo do uso preventivo de **const** na definição de tal ponteiro.

Comparando Ponteiros, Strings e Caracteres (Seção 9.4)

13. (a) Qual é a diferença entre `"A"` e `'A'`? (b) Qual é o tipo de `"A"`? (c) Qual é o tipo de `'A'`? (d) Quantos bytes ocupa `"A"`? (e) Quantos bytes ocupa `'A'`?

14. (a) Quais são as diferenças entre as definições das variáveis `str[]` e `ptr` a seguir? (b) Por que o item (a) desta questão usa `str[]` e não apenas `str`?

```
char str[] = "bola";
char *ptr ="bola";
```

15. Em que resulta a avaliação de cada uma das seguintes expressões?

(a) `"Bola"`
(b) `*"Bola"`
(c) `"Bola"[3]`
(d) `3["Bola"]`

16. Suponha que o cabeçalho `<stdio.h>` seja incluído em cada um dos seguintes programas. (a) O que há de errado com cada um deles? (b) Em quais situações o erro é detectado pelo compilador? (c) Em quais situações o erro poderá ocorrer em tempo de execução do programa?

(i)

```
int main(void)
{
   char *p = "bola";

   p[1] = 'a';
   printf("%s\n", p);

   p = "carro";
   printf("%s\n", p);

   return 0;
}
```

(ii)

```
int main(void)
{
   const char *p = "bola";

   p[1] = 'a';
   printf("%s\n", p);

   p = "carro";
   printf("%s\n", p);

   return 0;
}
```

(iii)

```
int main(void)
{
   char *const p = "bola";

   p[1] = 'a';
   printf("%s\n", p);

   p = "carro";
   printf("%s\n", p);

   return 0;
}
```

(iv)

```
int main(void)
{
   const char *const p = "bola";

   p[1] = 'a';
   printf("%s\n", p);

   p = "carro";
   printf("%s\n", p);

   return 0;
}
```

17. Suponha que `p` seja um ponteiro para o tipo **char**. Qual é o problema com cada uma das seguintes atribuições?
(a) `*p = "a";`
(b) `p = 'a';`

18. Por que a iniciação:

```
char *p = "bola";
```

é legal, mas a atribuição:

```
*p = "bola";
```

não o é.

19. (a) Por que a expressão `2["bola"]` é absolutamente legal em C? (b) Qual é o resultado dessa expressão?

20. (a) O que escreve na tela cada um dos seguintes programas? (b) Qual deles pode ser abortado? **Observação**: Suponha que cada um deles inclui `<stdio.h>`.

(i)

```
int main(void)
{
   char ar[] = {1, 2, 3, 4, 5, 7, 8, 9, 0},
        *p = ar;

   while (*p) {
      printf("%c\n", *p++);
   }

   return 0;
}
```

(ii)

```
int main(void)
{
   char ar[] = {'1', '2', '3', '4', '5', '7', '8', '9', '0'},
        *p = ar;

   while (*p) {
      printf("%c\n", *p++);
   }

   return 0;
}
```

(iii)

```
int main(void)
{
   char ar[] = "123457890",
       *p = ar;

   while (*p) {
      printf("%c\n", *p++);
   }
   return 0;
}
```

21. Qual é o resultado da avaliação de cada uma das seguintes expressões?
(a) `sizeof("Bola")`
(b) `strlen("Bola")`

22. Cite uma situação na qual o operador **sizeof** pode ser usado para determinar o tamanho de um string.

23. Dada a seguinte definição de variável:

```
char *semana[] = { "domingo", "segunda", "terca",
                   "quarta", "quinta", "sexta", "sabado" };
```

Qual é o significado de cada uma das expressões a seguir?
(a) `semana`
(b) `*semana`
(c) `**semana`
(d) `semana + 3`
(e) `*(semana + 3)`
(f) `*(*(semana + 3) + 2)`
(g) `semana[2]`

Funções de Biblioteca para Processamento de Strings (Seção 9.5)

24. Que facilidades oferece a função `LeString()`?

25. O que escreve na tela cada uma das seguintes chamadas de **printf**()? Justifique suas respostas.
(a) `printf("\nValor de '0': %d", '0');`
(b) `printf("\nValor de '\0': %d", '\0');`
(c) `printf("\nValor de '0': %c", '0');`
(d) `printf("\nValor de '\\0': %c", '\0');`

26. Um aluno de programação escreveu a seguinte função com o objetivo de concatenar dois strings, mas ela está incorreta. Descubra e corrija os erros de programação apresentados por esta função.

```
char *Concatena(char *str, const char *ptr)
{
   while (*str++)
      str++;

   while (*str++ = *ptr++)
      ; /* Instrução vazia */

   return str;
}
```

27. Explique o uso de **const** no protótipo da função **strlen**():

```
size_t strlen(const char *string)
```

28. Que cuidado deve ser tomado quando se usa o valor retornado por **strlen**()?

29. O seguinte programa aparece como exemplo num famoso livro programação em C. (a) Quais são os erros desse programa? (b) Qual desses erros é o mais grave?

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

char s1[] = "Bom ";
char s2[] = "dia";

void main(void)
{
   int p;

   p = strcat(s1, s2);
}
```

30. A função `CopiaString()`, apresentada a seguir, foi implementada com o objetivo de ser funcionalmente equivalente a **strcpy()**. No entanto, ela contém um erro. (a) Qual é esse erro? (b) Um compilador seria capaz de indicar esse erro?

```
char *CopiaString(char *destino, const char *origem)
{
   char *inicioStrDestino = destino; /* Guarda início do string destino */

      /* Copia cada caractere do string   */
      /* 'origem' para o string 'destino' */
   while (*origem++ = *++destino)
      ; /* Instrução vazia */

   return inicioStrDestino;
}
```

31. Se função **strcpy()** armazena o resultado da cópia do string recebido como segundo parâmetro no array recebido como primeiro parâmetro, para que serve o valor retornado por essa função?
32. A função **strcmp()** é útil em ordenação de strings? Explique.
33. Se a função **strcmp()** não é muito útil em ordenação de strings, para que ela serve afinal?
34. Em que diferem as funções **strcmp()** e **strcoll()**?
35. Em que situação chamadas de **strcoll()** produzem o mesmo efeito que chamadas de **strcmp()**?
36. O que é uma localidade em programação?
37. O que é colação de caracteres?
38. Para que serve a função **strstr()**?
39. Em que diferem as funções **strchr()** e **strrchr()**?
40. Como se faz um ponteiro `p` do tipo **char** * apontar para o caractere terminal de um string `str` utilizando apenas uma instrução?
41. Como se pode calcular o comprimento de um string representado pelo ponteiro `str` com uma única expressão usando **strchr()** [ou **strrchr()**]?
42. Descreva o funcionamento da função **strtok()**.
43. (a) Por que o primeiro parâmetro de **strtok()** não deve ser um string constante? (b) O que pode acontecer se essa recomendação não for seguida?
44. Que cuidado se deve tomar para evitar corrupção de memória quando se chama a função **strcat()**?
45. Em um livro de programação em C, uma função que calcula o comprimento de um string é implementada como:

```
int Comprimento(char string[])
{
   int i;

   for (i = 0; string[i] != '\0'; ++i)
      continue;

   return i;
}
```

(a) Qual é o problema com a declaração do parâmetro **string** dessa função? (b) Para que serve a instrução **continue** nessa função? (c) Reescreva essa função usando um melhor estilo (e conhecimento) de programação.

46. Por que o programa a seguir pode ser abortado?

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

int main(void)
{
   char *p, *str = "um dois tres";

   p = strtok(str, " ");

   while (p) {
      printf("Token: %s\n", p);
      p = strtok(NULL, " ");
   }
   return 0;
}
```

47. O que há de errado com o trecho de programa a seguir?

```
char primeiroNome[20] = "Maria",
     segundoNome[20] = "Jose",
     nomeCompleto[50];

   strcat(strcpy(nomeCompleto, primeiroNome), ' ');
   strcat(nomeCompleto, segundoNome);
```

48. (a) O que o programa a seguir exibe na tela? (b) Por que ele é executado indefinidamente?

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

int main(void)
{
   char *p, str[] = "um dois tres";

   p = strtok(str, " ");

   while (p) {
      printf("Token: %s\n", p);
      p = strtok(str, " ");
   }
   return 0;
}
```

49. Que operação sobre strings cada função a seguir implementa?

(a)

```
void F1(char *s1, const char *s2)
{
   while (*s1++)
      ; /* Vazio */

   for (--s1; *s1 = *s2; s1++, s2++ )
      ; /* Vazio */
}
```

(b)

```
int F2(const char *s)
{
   int c;

   for (c = 0; *s; s++, ++c)
      ; /* Vazio */

   return c;
}
```

(c)

```
int F3( const char *s1, const char *s2 )
{
   for ( ; *s1 && *s2; s1++, s2++ ) {
      if (*s1 != *s2) {
         break;
      }
   }
   return *s1 - *s2;
}
```

50. Assumindo que o programa a seguir inclui os cabeçalhos `<stdio.h>` e `<string.h>`, o que ele apresenta na tela?

```
int main( void )
{
   const char *s1 = "abcdefabcdef";
   const char *s2 = "def";

   printf( "\n%s\n", strstr(s1, s2) );

   return 0;
}
```

51. O que faz a função `F()` a seguir?

```
int F(const char str1[], const char str2[] )
{
   int i;

   for(i = 0; str1[i] == str2[i]; i++) {
      if( str1[i] == '\0' ) {
         return 1;
      }
   }

   return 0;
}
```

52. A função `F()` a seguir implementa, corretamente, a mesma comparação de strings efetuada por **strcmp()**. Explique como ela funciona.

```
int F(const char *str1, const char *str2)
{
   for(int i = 0; str1[i] == str2[i]; i++) {
      if(str1[i] == '\0') {
         return 0;
      }
   }

   return str1[i] - str2[i];
}
```

53. A função `F()` a seguir constitui outra forma de implementação de **strcmp()**. Explique seu funcionamento.

```
int F(const char *s1, const char *s2)
{
   for(; *s1 == *s2 && *s1 != '\0'; s1++, s2++)
      ;

   return *s1 - *s2;
}
```

54. Supondo que os cabeçalhos `<stdio.h>` e `<string.h>` sejam incluídos no programa no qual a função `F()` a seguir é definida, o que essa função faz?

```
int F(const char *s, int c)
{
   int n = 0;

   while((s = strchr(s, c)) != NULL) {
      s++;
      n++;
   }
   return n;
}
```

A Função main() (Seção 9.6)

55. (a) Considerando qualquer padrão da linguagem C, a função **main()** pode ter tipo de retorno **void**? (b) Se a resposta for negativa, por que existem programas que definem o tipo de retorno de **main()** como **void**?

56. (a) Quais são os significados dos parâmetros `argc` e `argv` usados pela função **main()**? (b) Quais são as origens das denominações *argc* e *argv*? (c) Esses parâmetros precisam ser realmente denominados assim?

57. Como um programa pode obter seu nome de arquivo executável?

58. Suponha que um programa precisa processar dois argumentos de linha de comando. Que teste ele deve efetuar ao início de sua execução para verificar se foi invocado corretamente?

Classificação e Transformação de Caracteres (Seção 9.7)

59. Qual é o propósito geral do cabeçalho `<ctype.h>`?

60. Que função declarada no cabeçalho `<ctype.h>` você utilizaria num programa para testar se um caractere é classificado como:

(a) Alfanumérico
(b) Letra
(c) Dígito
(d) Letra maiúscula

61. O que faz a seguinte função `F()`?

```
int F(const char *str)
{
   for (; *str; str++)
      if (!isalnum(*str))
         return 0;

   return 1;
}
```

62. Que função declarada em `<ctype.h>` é usada para converter letras maiúsculas em minúsculas?
63. Qual será o retorno da seguinte chamada da função **isalpha**(): `isalpha('ã')`?

Ordem de Avaliação de Parâmetros (Seção 9.8)

64. Por que não se deve fazer suposições sobre a ordem de avaliação de parâmetros numa chamada de função?
65. Dependendo do compilador utilizado, o programa a seguir:

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

void ExibeInts(int x, int y)
{
   printf("\n\tPrimeiro valor: %d", x);
   printf("\n\tSegundo valor: %d", y);
}

int main(void)
{
   int  x = 10;

   ExibeInts( x, ++x );
   putchar('\n');

   return 0;
}
```

pode produzir dois resultados possíveis:

```
        Primeiro valor: 11
        Segundo valor: 11
```

ou:

```
        Primeiro valor: 10
        Segundo valor: 11
```

Explique por que.

66. Considere os seguintes protótipos de função:

```
void F1(char *p)
void F2(const char *p)
void F3(char **p)
```

Considere ainda o array `ar[]` definido como:

```
char ar[20] = "bola";
```

(a) Que funções poderiam ser chamadas usando o array `ar[]` e como seriam essas possíveis chamadas?
(b) Em quais chamadas permitidas o conteúdo do array `ar[]` pode ser alterado?

67. Considere os mesmos protótipos de função do exercício anterior e o ponteiro `str` definido como:

```
char *str = "bola";
```

(a) Que funções poderiam ser chamadas usando o ponteiro `str` e como seriam essas possíveis chamadas?
(b) Dentre as chamadas permitidas, quais podem causar aborto de um programa que as execute?
(c) Em quais chamadas permitidas o valor de `str` pode ser alterado?

68. Considere os mesmos protótipos de função do penúltimo exercício e o ponteiro `str2` definido como:

```
const char *str2 = "bola";
```

(a) Que funções poderiam ser chamadas usando o ponteiro `str2` e como seriam essas possíveis chamadas?
(b) Dentre as chamadas permitidas, quais podem causar aborto de um programa que as execute?
(c) Em quais chamadas permitidas o valor de `str2` pode ser alterado?

69. Suponha que uma função `F()` possua o seguinte protótipo:

```
void F(const char *p)
```

Considerando que o ponteiro `str` seja definido como:

```
char *str = "bola";
```

A chamada de `F()` a seguir é legal?

```
F(str);
```

70. Suponha que uma função `F()` possua o seguinte protótipo:

```
void F(char *p)
```

Considerando que o ponteiro `str` seja definido como:

```
const char *str = "bola";
```

A chamada de `F()` a seguir é legal?

```
F(str);
```

71. (a) O que o seguinte programa exibe na tela? (b) Por que esse programa é abortado? (c) Por que o compilador é incapaz de apresentar qualquer mensagem de advertência?

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

void EscreveStrSemEspacos(const char *str)
{
   char *p;

   while (isspace(*str)) {
      ++str;
   }

   p = strchr(str, '\0') - 1;

   while (isspace(*p)) {
      --p;
   }

   *++p = '\0';

   printf("\n\"%s\"\n", str);
}

int main(void)
{
   char *str1 = "    Time e'     ",
        str2[] = "    Botafogo     ";

   EscreveStrSemEspacos(str1);
   EscreveStrSemEspacos(str2);

   return 0;
}
```

Conversão de Strings Numéricos em Números (Seção 9.9)

72. Descreva o funcionamento da função **atoi()**.
73. Em qual situação o resultado retornado por **atoi()** é ambíguo e por quê?
74. Descreva o funcionamento da função **strtod()**.
75. (a) Para quer serve o último parâmetro de **strtod()**? (b) Como esse parâmetro deve ser usado?
76. Por que, quando o segundo parâmetro de **strtod()** é **NULL**, o valor retornado por essa função pode ser ambíguo?

9.12 Exercícios de Programação

9.12.1 Fácil

EP9.1 Escreva uma função, denominada `TransformaStr()`, que recebe um string como primeiro parâmetro e um caractere como segundo parâmetro. Quando o segundo parâmetro for o caractere `'M'`, essa função deve transformar o string de tal modo que todas as suas letras passem a ser maiúsculas. Quando o segundo parâmetro for `'m'`, a função deve transformar o string de tal modo que todas as suas letras sejam minúsculas. Quando o segundo parâmetro não for `'M'` ou `'m'` essa função não deve promover nenhuma transformação no string. A função deverá retornar o endereço do string transformado. [**Sugestão**: Utilize as funções **tolower()** e **toupper()** discutidas na **Seção 9.7.2**.]

EP9.2 Escreva um programa, semelhante àquele apresentado como exemplo na **Seção 9.10.8**, que lê e verifica a validade de uma data no formato ISO 8601 (i.e., no formato: `aaaa/mm/dd`). [**Sugestão**: Estude o exemplo apresentado na **Seção 9.10.8** e descubra o que precisa ser alterado naquele programa para obter a solução para o problema corrente.]

EP9.3 Implemente uma função, denominada `ComparaStrings()`, funcionalmente equivalente à função **strcmp()**. [**Sugestão**: Use um laço de repetição para comparar os strings caractere a caractere. Se for encontrada uma diferença entre respectivos caracteres ou for encontrado o caractere terminal de um dos strings encerre o laço e retorne a diferença entre os dois últimos caracteres acessados.]

EP9.4 Implemente uma função, denominada `EncontraPrimeiroChar()`, funcionalmente equivalente à função **strchr()**. [**Sugestão**: Utilize um laço de repetição para comparar cada caractere do string com o caractere procurado. Esse laço deve encerrar quando o caractere procurado ou o caractere terminal for encontrado. No primeiro caso, a função retorna o endereço do caractere, enquanto, no segundo caso, ela retorna **NULL**. Lembre-se que o caractere procurado pode ser o caractere terminal do string.]

EP9.5 Implemente uma função, denominada `EncontraUltimoChar()`, funcionalmente equivalente à função **strrchr()**. [**Sugestão**: Faça um ponteiro local à função apontar para o caractere terminal do string usando **strchr()**. Então, use um laço de repetição que decremente esse ponteiro até que ele aponte para o caractere procurado ou seja menor do que o ponteiro que aponta para o início do string. No primeiro caso, a função retorna o valor do ponteiro auxiliar, enquanto, no segundo caso, ela retorna **NULL**.]

EP9.6 Escreva uma função que retorna `1` quando um string possui apenas letras e dígitos ou `0`, em caso contrário. [**Sugestão**: Utilize a função **isalnum()** discutida na **Seção 9.7.1**.]

EP9.7 Escreva uma função que substitui cada caractere de tabulação de um string por um espaço em branco. [**Sugestão**: Utilize **strchr()** para localizar cada caractere de tabulação `'\t'` e então substitua-o.]

EP9.8 (a) Escreva uma função, denominada `OcorrenciasCar()`, que conta o número de ocorrências de um caractere num string. [**Sugestões**: (1) Use uma variável local para contar o número de ocorrências solicitado. Essa variável deve ser iniciada com zero. (2) Use um laço **while** cuja condição de parada seja o fato de o parâmetro que representa o string apontar para o caractere terminal desse string. (3)

No corpo do laço, verifique se esse parâmetro aponta para o caractere procurado e, se for o caso, incremente a variável que armazena o número de ocorrências. Em seguida, incremente o parâmetro. (4) Após o final do laço, retorne o valor da variável que conta as ocorrências.] (b) Escreva um programa que lê strings e caracteres isolados via teclado e informa o número de ocorrências de cada caractere no respectivo string. O programa deve encerrar quando o usuário introduzir um string vazio. [**Sugestão**: Use `LeString()` e `LeCaractere()` da biblioteca LeituraFacil para ler os strings e os caracteres, respectivamente.]

EP9.9 (a) Escreva uma função que substitui todas as ocorrências de um dado caractere num string por outro caractere. O protótipo dessa função deve ser:

```
char *SubstituiCaracteres(char *str, int substituir, int novo)
```

Os parâmetros dessa função são interpretados como:

- `str` é o string no qual serão feitas as substituições
- `substituir` é o caractere que será substituído
- `novo` é o caractere que substituirá as ocorrências do segundo parâmetro

O retorno dessa função deve ser o endereço do string recebido como parâmetro. (b) Escreva um programa para testar a função `SubstituiCaracteres()`. [**Sugestão**: Utilize como base a função `OcorrenciasCar()` solicitada no exercício **EP9.8**. Então, em vez de contar as ocorrências de um dado caractere, você as substituirá.]

EP9.10 Escreva uma função que remove todos caracteres que não são letras de um string. [**Sugestões**: (1) Use a função **isalpha()** discutida na **Seção 9.7.1**. (2) Use as sugestões apresentadas para o exercício **EP9.9**.]

EP9.11 (a) Escreva uma função que retorna o token de ordem `n` de um string, se este existir; caso contrário, a função deve retornar **NULL**. O protótipo dessa função deve ser:

```
char *EnesimoToken(char *str, const char *separadores, int n)
```

Nesse protótipo, os parâmetros têm os seguintes significados:

- `str` é o string no qual o enésimo token será procurado
- `separadores` é um string contendo os possíveis separadores de tokens
- `n` é o número de ordem do token desejado

(b) Escreva um programa para testar a função `EnesimoToken()`.

[**Sugestão**: Utilize a função **strtok()** para ler e descartar `n - 1` tokens do string. Se **strtok()** retornar **NULL** antes, não existe enésimo token. Caso contrário, chame e retorne o valor retornado por **strtok()**.]

EP9.12 (a) Escreva uma função que copia os `n` caracteres iniciais de um string. O protótipo dessa função deve ser:

```
char *CopiaInicio(char *destino, const char *origem, int n)
```

Os parâmetros dessa função são interpretados como:

- `destino` é o array que receberá a cópia
- `origem` é o string que doará os caracteres
- `n` é o número de caracteres iniciais que serão copiados.

O retorno dessa função deve ser o endereço do array recebido como primeiro parâmetro. (b) Escreva um programa para testar a função `CopiaInicio()`. [**Observação**: A função `CopiaInicio()` é semelhante à função **strncpy()** da biblioteca padrão de C.]

EP9.13 (a) Escreva uma função que copia os `n` caracteres finais de um string. O protótipo dessa função deve ser:

```
char *CopiaFinal(char *destino, const char *origem,  int n)
```

Nesse protótipo, os parâmetros têm os seguintes significados:

- `destino` é o array que receberá a cópia
- `origem` é o string que doará os caracteres
- `n` é o número de caracteres finais que serão copiados, sem incluir o caractere terminal do string `origem`.

O retorno dessa função deve ser o endereço do array recebido como parâmetro. (b) Escreva um programa para testar a função `CopiaFinal()`. [**Sugestão**: Utilize um ponteiro `p` local à função e faça-o apontar para o primeiro caractere a ser copiado e, então, chame **strcpy()** como `strcpy(destino, p)`.]

EP9.14 (a) Escreva uma função que copia `n` caracteres de um string a partir de uma dada posição. O protótipo dessa função deve ser:

```
char *CopiaNCaracteres(char *destino, const char *origem, int pos, int n)
```

As interpretações dos parâmetros nesse protótipo são as seguintes:

- `destino` é o array que receberá a cópia
- `origem` é o string que fornecerá os caracteres
- `pos` é a posição do string `origem` a partir da qual os caracteres serão copiados
- `n` é o número de caracteres que serão copiados.

O retorno dessa função deve ser o endereço do array recebido como parâmetro. (b) Escreva um programa para testar a função `CopiaNCaracteres()`. [**Sugestão**: Utilize a função `CopiaInicio()` solicitada no exercício **EP9.12**.]

EP9.15 Escreva um programa que verifica se um número de CPF introduzido pelo usuário é válido. Um número de CPF tem 11 dígitos divididos em duas partes: a parte principal com 9 dígitos e os dígitos verificadores, que são os dois últimos dígitos. A validação de um número de CPF segue o procedimento descrito abaixo:

- Verificação do primeiro dígito de controle (penúltimo dígito do número):
  - Multiplique os inteiros representados pelos dígitos da parte principal, do primeiro ao último, respectivamente, por `10`, `9`, `8`, ..., `2` e some os resultados obtidos.
  - Calcule o resto da divisão do resultado obtido no item anterior por `11`.
  - Se o resto da divisão for `0` ou `1`, o primeiro dígito verificador deverá ser igual a `0`; caso contrário, esse dígito deverá ser igual a `11` menos o referido resto de divisão.
- Verificação do segundo dígito de controle (último dígito do número):
  - Multiplique os inteiros representados pelos dígitos da parte principal e pelo primeiro dígito de controle, do primeiro ao último, respectivamente, por `11`, `10`, `9`, ..., `2` e some os resultados obtidos.
  - Calcule o resto da divisão do resultado obtido no item anterior por `11`.
  - Se o resto da divisão for `0` ou `1`, o segundo dígito verificador deverá ser igual a `0`; caso contrário, esse dígito deverá ser igual a `11` menos o último resto de divisão.

A principal diferença entre essas duas verificações é que a segunda inclui o primeiro dígito de controle.

[**Sugestão**: Esse problema é semelhante àquele de verificação de número de PIS/PASEP apresentado como exemplo na **Seção 9.10.3**.]

EP9.16 Escreva um programa semelhante àquele apresentado na **Seção 9.10.10**, denominado `multiplicaints` (ou `multiplicaints.exe`) que multiplica os argumentos de linha de comando que acompanham o nome do programa se eles forem todos números inteiros.

EP9.17 Escreva um programa semelhante àquele apresentado na **Seção 9.10.10**, denominado `somareais` (ou `somareais.exe`), que soma números reais passados para o programa como argumentos de linha de comando. [**Sugestão**: Use a função **strtod()**, discutida na **Seção 9.9.2**, para converter os parâmetros do programa em números reais.]

EP9.18 Escreva um programa que recebe um valor inteiro positivo `N` como argumento de linha de comando e apresenta como resultado a sequência de Fibonacci que contém `N` termos. Se `N` não for um valor válido para o programa ou estiver ausente, o programa deve responder adequadamente. [**Sugestão**: Use como modelos os exemplos apresentados na **Seção 9.10.10** e na **Seção 7.6.7**.]

EP9.19 Escreva um programa que recebe um valor inteiro positivo `N` como argumento de linha de comando e informa se `N` faz parte de alguma sequência de Fibonacci. Se `N` não for um valor válido para o programa ou estiver ausente, o programa deve responder adequadamente. [**Sugestão**: Siga a sugestão do exercício anterior.]

EP9.20 Escreva um programa que exibe na tela a segunda metade de um string. [**Sugestões**: (1) Calcule a metade do tamanho do string usando **strlen()**. (2) Faça um ponteiro apontar para essa posição. (3) Use esse ponteiro com uma chamada de **printf()**.]

EP9.21 Escreva um programa que exibe na tela a primeira metade de um string. [**Sugestões**: (1) Encontre a posição central do string usando **strlen()**. (2) Use um laço de contagem que chame **putchar()** para exibir cada caractere do string do seu início até a sua posição central.]

EP9.22 (a) Escreva uma função que acrescenta um caractere ao final de um string. O protótipo dessa função deve ser:

```
char *AcrescentaCaractere(char *str, int c, size_t tam)
```

Os parâmetros dessa função são interpretados como:

- `str` é o string que terá um caractere acrescentado
- `c` é o caractere que será acrescentado
- `tam` é o tamanho do array que contém o string.

O retorno dessa função deve ser o endereço do string alterado, se for possível acrescentar o caractere ou **NULL**, se não houver espaço suficiente. (b) Escreva um programa que lê strings e caracteres isolados via teclado, tenta acrescentar cada caractere no respectivo string e informa o resultado da operação. O programa deve encerrar quando o usuário digitar apenas `[ENTER]` quando instado a introduzir um string. [**Sugestões**: (1) Use **strlen()** para checar se há espaço suficiente para acréscimo de um novo caractere. (2) Se houver espaço para acréscimo, use, por exemplo, **strlen()** ou **strchr()** para acessar o local da inserção, que deve ser a posição corrente do caractere `'\0'`. (3) Não esqueça de acrescentar um novo caractere terminal ao string.]

EP9.23 Escreva um programa que apresenta na tela uma frase (string) introduzida pelo usuário em forma de escada. Isto é, cada palavra constituinte da frase é exibida numa linha separada e endentada em relação à palavra anterior, como por exemplo:

```
Isto
    e'
      um
        teste
```

[**Sugestão**: Use a **strtok()** para extrair cada palavra da frase e a função **strlen()** para calcular a endentação de uma palavra em relação àquela exibida na linha anterior.]

EP9.24 (a) Escreva uma função, denominada `InverteString()`, que copia um string invertido (segundo parâmetro) para um array (primeiro parâmetro). (b) Escreva um programa que lê strings via teclado e apresenta-os invertidos na tela. [**Sugestão**: Defina um ponteiro `p` local à função `InverteString()` e faça-o apontar para o último caractere do string usando **strchr()**. Então, use um laço de repetição para copiar cada caractere correntemente por `p` para cada elemento do array e decrementar esse ponteiro. O laço deve encerrar quando `p` apontar para o endereço inicial do string.]

EP9.25 (a) Escreva uma função, denominada `EhVogal()`, que verifica se o caractere recebido como parâmetro é vogal. [**Sugestão**: Use **strchr()** para verificar se o caractere faz parte do string constante `"aeiouAEIOU"`.] (b) Escreva uma função, denominada `EhConsoante()`, que verifica se um caractere é consoante. [**Sugestão**: Use **isalpha()** e `EhVogal()`.] (c) Escreva um programa que lê uma palavra via teclado e informa quantas vogais e consoantes a palavra possui. [**Sugestão**: Use a função `LeNome()`, definida na **Seção 9.10.1** e as funções solicitadas nos itens (a) e (b).]

EP9.26 Escreva um programa que lê um número positivo menor do que `5000` e apresenta na tela o número correspondente usando algarismos romanos. Esse programa deve ser funcionalmente equivalente àquele solicitado no exercício **EP5.18**, mas deve usar os arrays de strings: `unidades[]`, `dezenas[]`, `centenas[]` e `milhares[]` para armazenar os strings constantes que correspondem, respectivamente, à possível unidade, dezena, centena e milhar do número lido.

EP9.27 (a) Escreva uma função, denominada `OcorrenciasStr()`, que conta o número de ocorrências de um string (primeiro parâmetro) em outro string (segundo parâmetro). [**Sugestões**: (1) Defina uma variável de contagem e inicie-a com `0`. (2) Crie um laço de repetição infinito no corpo do qual a função **strstr()** é chamada tendo o string a ser procurado como segundo parâmetro. (3) Na primeira execução do corpo do laço, o primeiro parâmetro de **strstr()** deve ser o primeiro parâmetro da função que está sendo implementada. Nas execuções subsequentes do corpo do laço, esse parâmetro de **strstr()** deve ser acrescido do tamanho do string procurado. (4) Esse laço deve encerrar quando **strstr()** retornar **NULL**.] (b) Escreva um programa para testar a função solicitada no item (a).

EP9.28 A função `LeNome()`, apresentada na **Seção 9.10.1**, não testa se o string lido é constituído apenas por espaços em branco, de forma que o usuário pode digitar um nome que será invisível quando exibido. Reescreva essa função de maneira a corrigir esse defeito. [**Sugestão**: Use a função `RemoveBrancosInicio()`, definida na **Seção 9.10.7**, para remover eventuais espaços em branco no início do string lido. Então teste se o string se torna vazio após a chamada dessa função. Se esse for o caso, inste o usuário a introduzir um novo nome.]

9.12.2 Moderado

EP9.29 Implemente uma função, denominada `PosicaoEmString()`, funcionalmente equivalente à função **strstr()**.

EP9.30 Escreva um programa que apresenta todos os anagramas que podem ser formados com as letras de uma palavra introduzida pelo usuário via teclado. **NB**: Um **anagrama** é o resultado do rearranjo das letras de uma palavra que resulta em outra palavra, utilizando cada letras da referida palavra uma única vez. [**Sugestões**: (1) Escreva uma função que lê strings contendo apenas letras. (2) Utilize o método de geração de permutações por ordenação lexicográfica discutido na **Seção 8.11.7** para gerar possíveis anagramas de um string constituído apenas por letras.]

EP9.31 (a) Escreva uma função que remove de um string todas as ocorrências de um dado caractere. O protótipo dessa função deve ser:

```
char *RemoveCaractere(char *str, int remover)
```

Nesse protótipo, `str` é o string que será eventualmente modificado, `remover` é o caractere a ser removido e o retorno da função deve ser o endereço inicial do string. [**Sugestões**: (1) Use dois ponteiros locais à função, denominados `p` e `inicio` e faça-os apontar para string `str` recebido como parâmetro. (2) Use um laço **while** que encerre quando `str` apontar para o caractere terminal do string. No corpo desse laço, copie para o endereço apontado por `p` qualquer caractere que não seja igual ao caractere que será removido e faça `p` e `str` apontarem para um caractere adiante. (3) Depois do laço, acrescente um caractere terminal após o último caractere apontado por `p`. (4) Retorne o valor do ponteiro `inicio`.] (b) Escreva um programa que lê um string e um caractere via teclado e remove todas as ocorrências do caractere no string. O string deve ser apresentado na tela antes e depois das eventuais substituições.

EP9.32 (a) Implemente uma função, cujo protótipo é:

```
char *RemoveCaracteres(char *str, const char *aRemover)
```

que remove do primeiro string recebido como parâmetro todos os caracteres presentes no segundo parâmetro, que também é um string. O retorno dessa função deve ser o endereço do string eventualmente alterado. (b) Escreva uma função **main()** que recebe dois strings como argumentos de linha de comando e remove do primeiro string todos os caracteres presentes no segundo string. [**Sugestão**: Use a função `RemoveCaractere()` solicitada no exercício **EP9.31**.]

EP9.33 (a) Escreva uma função que remove todas as ocorrências de um dado string em outro string e retorna o número de remoções efetuadas. [**Sugestões**: (1) defina três variáveis: `p`, usada como ponteiro auxiliar; `tamSubstring`, que armazenará o tamanho do substring que será removido e `nRemocoes`, que armazenará o número de remoções. (2) Calcule o tamanho do substring que será removido e atribua-o a `tamSubstring`. (3) Use um laço **while** que encerra quando o parâmetro que representa o string é **NULL** ou aponta para o caractere `'\0'`. (4) No corpo desse laço, use **strstr()** para encontrar a próxima ocorrência do substring no string e atribua o retorno dessa função a `p`. (5) Se `p` for **NULL**, encerre o laço. Caso contrário, copie para o array apontado por `p` o string que começa em `p + tamSubstring` e incremente a variável `nRemocoes`. (6) Ainda no corpo do laço, atribua `p` ao parâmetro que representa o string.] (b) Escreva um programa que lê dois strings via teclado, remove as ocorrências do segundo string no primeiro e apresenta o resultado da operação.

EP9.34 (a) Escreva uma função que substitui todas as ocorrências de um substring num string por outro substring. [**Sugestões**: (1) Use **strlen()** para calcular os tamanhos dos dois substrings. (2) Use um laço de repetição que encerra quando a função **strstr()** indicar que não há mais ocorrências do substring a ser substituído. (3) No corpo desse laço, determine o espaço para o qual serão copiados os caracteres substitutos, discriminando as substituições em duas categorias, dependendo dos tamanhos dos dois substrings. Isto é, se o tamanho do substring que será substituído for maior do que o daquele que o substituirá, devem-se mover caracteres para trás; caso contrário, devem-se mover caracteres para frente. Nos dois casos, o deslocamento de caracteres deve ser igual à diferença de tamanho entre os dois substrings. (4) Copie os caracteres do substring para o espaço determinado no passo anterior.] (b) Escreva um programa para testar a função especificada em (a). [**Sugestão**: Use a função `OcorrenciasStr()`, solicitada no exercício **EP9.27**, para calcular o tamanho que o string resultante das substituições terá quando a operação estiver concluída e determinar se o array que armazenará o resultado terá espaço suficiente para contê-lo.]

EP9.35 (a) Escreva uma função, denominada `IntEmString()`, que converte um valor do tipo **int** em string. [**Sugestões**: (1) Defina um array de duração fixa local à função para armazenar o resultado da operação.

O tamanho desse array é o valor de uma constante simbólica do programa. É vital que o referido array tenha duração fixa, pois, caso contrário, ele seria considerado um zumbi (v. **Seção 8.9.4**). (2) Verifique se o número a ser convertido é negativo e, se for o caso, armazene essa informação numa variável local e considere o valor absoluto do número para conversão. Ao final da conversão, se o número for negativo, o sinal de menos será acrescentado ao string. (3) Para evitar que o string que conterá os dígitos que compõem o número precise ser invertido ao final do processo, armazene-os do final para o início do array. Portanto o primeiro passo para obter o resultado desejado é armazenar o caractere terminal do string na última posição do array. (4) Use um laço **do-while** para extrair e armazenar no array cada dígito que compõe o número. Enquanto isso é efetuado, conte quantos caracteres estão sendo armazenados e compare esse valor com o tamanho do array para evitar corrupção de memória; i.e., se a quantidade de caracteres (dígitos, sinal e caractere terminal) exceder a capacidade de armazenamento do array, retorne **NULL**, indicando que a conversão não foi bem sucedida. (5) Finalmente, se o número for negativo e ainda houver espaço no array, acrescente o sinal de menos ao string que contém o resultado e retorne o endereço inicial do string (e não do array que o armazena).] (b) Escreva um programa para testar a função especificada no item (a).

EP9.36 (a) Escreva uma função, denominada `DoubleEmString()`, que converte um valor do tipo **double** em string. Essa função deve truncar a parte fracionária na segunda casa decimal. [**Sugestões**: (1) Separe o número em partes inteira e fracionária, conforme ensinado na **Seção 7.5**. (2) Armazene as partes inteira e fracionária no array que conterá o resultado como faz a função `IntEmString()` solicitada no exercício **EP9.35**. (3) Não esqueça que existe um ponto decimal separando as duas partes.] (b) Escreva um programa que testa a função especificada no item (a).

EP9.37 (a) Escreva uma função que separa um string em tokens, como faz a função **strtok()** da biblioteca padrão de C. (b) Escreva um programa que testa a função solicitada no item (a) e compara os resultados obtidos por meio dessa função com aqueles obtidos via **strtok()**. [**Sugestões para o item (a)**: (1) Use um ponteiro de duração fixa local à função, denominado `proximoToken`, para armazenar o endereço do primeiro caractere do próximo token do string recebido como parâmetro. Defina ainda os seguintes ponteiros locais: `s`, que apontará para o string no qual a busca pelo token será efetuada, e `inicio`, que guarda o início do token corrente. (2) Quando o primeiro parâmetro da função em discussão não for **NULL**, a busca pelo próximo token começa no endereço indicado por esse parâmetro. Caso contrário, a busca pelo próximo token começa no endereço armazenado na variável `proximoToken`, a não ser que essa variável também seja **NULL**. Nesse último caso, não há mais token a ser encontrado e a função retorna **NULL**. (3) Quando há possíveis tokens a serem encontrados, saltam-se eventuais separadores (especificados no segundo parâmetro) que se encontrem no início do string no qual a busca será realizada. Se, durante essa operação, o final do string for atingido, não há mais token no string sendo processado e a função retorna `''`. Ainda nesse caso, a variável `proximoToken` recebe o valor **NULL**, de forma que a próxima chamada da função tendo **NULL** como primeiro parâmetro não procurará um novo token. (4) Se, após saltar os separadores iniciais, o final do string não for atingido, haverá pelo menos mais um token no string. Então, guarda-se o endereço desse token que será retornado na variável `inicio` e procura-se o final desse token (i.e., um separador de token especificado no segundo parâmetro ou `'\0'`) usando a variável `s`. Quando um separador é encontrado, ele é substituído pelo caractere terminal `'\0'` e à variável `proximoToken` é atribuído o endereço do caractere que segue esse separador. Se, nesse passo não for encontrado nenhum separador, não haverá mais token na próxima chamada da função, e, assim, à variável `proximoToken` é atribuído **NULL**. (5) A última instrução da função retorna o endereço do token encontrado, que foi armazenado na variável `inicio`.]