CAPÍTULO 8
STRINGS

8.1 Introdução

Strings são **cadeias** (ou **seqüências**) **de caracteres**. Em C, um ***string*** consiste em um array de caracteres terminado pelo **caractere nulo**, representado pela seqüência de escape `'\0'`. Um ***string*** constante é qualquer seqüência de caracteres entre aspas, e seu tipo de dados é um array de elementos do tipo **char**. O compilador sempre acrescenta o caractere nulo para designar o final de um ***string*** constante.

Strings constituem um tipo de dados essencial para qualquer programa interativo. Mesmo programas que possuem interfaces gráficas do usuário sofisticadas precisam ler, processar e imprimir ***strings***. Por isso, a maioria das linguagens de programação de alto nível oferece inúmeras operações predefinidas de manipulação de ***strings***. Em C, operações para processamento de strings são implementadas como funções que fazem parte do módulo string da biblioteca padrão da linguagem. As funções de biblioteca para processamento de ***strings*** mais comumente utilizadas são aqui apresentadas, enquanto outras funções desta categoria são exploradas no **Volume II**.

8.2 Definições e Iniciações de Strings

Pode-se armazenar um ***string*** em memória utilizando-se um array de elementos do tipo **char**. Pode-se ainda iniciar um array de caracteres com um ***string*** constante. Por exemplo:

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

Devido à onipresença do caractere terminal do ***string***, quando não é especificado explicitamente, o tamanho do array que o armazena é sempre um a mais do que o número de caracteres visíveis no ***string***. Assim, o array `str` do exemplo anterior possui tamanho igual a 19 (18 caracteres visíveis mais o caractere terminal).

Se o tamanho do array for especificado, este tamanho deve ser, no mínimo, igual ao número de caracteres presentes na iniciação; os elementos remanescentes no array, se for o caso, serão iniciados com 0. Por exemplo, na definição a seguir:

```
char  str[10] = "ola";
```

`str[0]` assume o valor `'o'`, `str[1]` assume o valor `'l'`, `str[2]` assume o valor `'a'` e `str[3]` assume o valor `'\0'`. Os elementos restantes (i.e., de `str[4]` a `str[9]`) recebem o valor 0. Em outras palavras, o compilador interpreta esta última iniciação como:

```
char  str[10] = {'o', 'l', 'a', '\0'};
```

Por causa disso, a definição:

```
char  str[4] = "Tudo bem.";
```

seria considerada ilegal.

É interessante notar, entretanto, que a iniciação a seguir:

```
char ar[4] = "Bola";
```

é legal, mas, neste caso, o caractere terminal de ***string*** não é armazenado no array `ar`. Portanto, o array `ar` contém caracteres, mas não contém um ***string*** de acordo com a definição apresentada acima.

Em C, ainda é permitida a iniciação de um ponteiro para **char** com um ***string*** constante. Por exemplo:

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

Entretanto, esta última definição é ligeiramente diferente das definições precedentes que utilizam arrays. Uma diferença entre esta última definição e a definição:

```
char  str[] = "Isto 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 de `ptr` pode ser modificado, enquanto o endereço `str` não o pode (v. **Seção 3.2**). Note, entretanto, que se o valor de `ptr` for modificado, o endereço com o qual este ponteiro foi iniciado será perdido (i.e., o ***string*** para onde `ptr` estava apontando não mais poderá ser acessado).

A razão pela qual se pode iniciar um ponteiro para **char** com um ***string*** constante é que um ***string*** é um array de caracteres, de modo que um ***string*** constante representa um ponteiro para o primeiro caractere do array de caracteres que contém o ***string***. Conseqüentemente, pode-se atribuir um ***string*** constante a um ponteiro que aponte para um **char**.

Deve-se salientar que, apesar da aparência, o lado direito da iniciação de um array de caracteres não é realmente um ***string*** constante. Por exemplo, na iniciação:

```
char  str[10] = "ola";
```

`"ola"` não representa um ***string*** constante, como ocorre no caso da iniciação de um ponteiro para **char**, conforme visto acima.

Uma importante diferença entre as iniciações do array `str` e do ponteiro `ptr` acima é o fato de alguns compiladores armazenarem ***strings*** constantes em posições de memória cujos conteúdos não podem ser modificados (i.e., são apenas para leitura). Em tal situação, qualquer tentativa de modificar o ***string*** para onde o ponteiro `ptr` aponta gera um erro de execução do programa. Para não correr riscos, considere ***realmente constantes*** os conteúdos de ***strings*** cujos endereços são atribuídos a ponteiros.

Os exemplos apresentados a seguir chamam a atenção para algumas dúvidas que um programador inexperiente em C pode ter com o uso de ponteiros, arrays e ***strings***.

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

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

Esta instrução é ***ilegal*** porque `ar` representa o endereço do elemento inicial do array `ar` e este endereço não pode ser modificado.

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

Esta instrução é ***legal***, ela representa simplesmente a atribuição do caractere `'a'` ao terceiro elemento do array `ar`, que é um array de caracteres.

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

Esta instrução é ***legal*** devido à relação entre ponteiros e arrays (v. **Seção 7.4**); esta atribuição modifica o valor do elemento de índice 5 do ***string*** `"10 esbacos"` para `'b'`, de modo que este ***string*** se torna `"10 espacos"`. Note que o valor do ponteiro `ptr` em si ***não é modificado***. ***Importante***: apesar de ser legal, esta instrução pode causar sérios problemas se o ***string*** for armazenado numa área de memória apenas para leitura. Portanto, é melhor evitar instruções que tentem modificar ***strings*** considerados constantes.

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

Esta instrução é ***legal*** e é exatamente eqüivalente à instrução anterior (e também pode causar o mesmo problema que aquela).

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

Esta instrução é obviamente legal. Talvez, menos óbvio seja o fato de `ptr` agora apontar para outra posição de memória, que é aquela onde 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';
```

Como visto anteriormente, esta instrução é sintaticamente legal. Entretanto, agora, esta instrução irá provavelmente causar problemas quando for executada. O problema é que, devido à atribuição no item anterior, `ptr` aponta para o ***string*** `"OK"`, que possui apenas três posições de memória alocadas, e esta última instrução atribui o valor `'b'` à terceira posição de memória além do final do ***string*** `"OK"`. Portanto, está se modificando uma posição de memória que não está alocada para este string. Em outras palavras, esta instrução não irá causar um erro de compilação, mas certamente causará um erro durante a execução do programa.

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

Esta instrução não é ilegal, mas provavelmente é desprovida de significado. Aqui, está tentando-se atribuir um valor para o caractere para o qual `ptr` aponta. O ***string*** constante `"ilegal?"` é interpretado como o endereço do elemento inicial do array que contém este ***string*** em memória. Portanto, esta atribuição representa, na realidade, a tentativa de atribuição de um endereço a uma variável do tipo **char**. Neste caso, o padrão ISO requer apenas que o compilador emita uma mensagem de advertência (v. **Seção 3.2.5**).

```
putchar(3["Estranho"]);
```

Por mais estranho que possa parecer, esta instrução é perfeitamente legal e imprime o caractere `'r'` no meio de saída padrão. Para entender por que a expressão `3["Estranho"]` é legal, note que, de acordo com a relação entre ponteiros e arrays vista na **Seção 7.4**, esta expressão é o mesmo que `3 + "Estranho"`, já que o ***string*** constante `"Estranho"` é avaliado como um ponteiro para o local onde este ***string*** é armazenado, conforme foi visto na presente seção. Ora, mas a soma é uma operação comutativa; portanto, `3 + "Estranho"` é o mesmo que `"Estranho" + 3` e esta última expressão é, novamente usando a relação apresentada na **Seção 7.4**, o mesmo que `"Estranho"[3]`. Esta última expressão é evidentemente legal e representa o caractere de índice 3 do ***string*** `"Estranho"`.

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

8.3 Comparação entre Ponteiros, Strings e Caracteres

As diferenças entre ***strings*** constantes contendo apenas um caractere visível e caracteres constantes é uma causa comum de confusão entre iniciantes em C. Uma destas diferenças refere-se ao espaço ocupado por um caractere constante e um ***string*** constante consistido em apenas um caractere: no primeiro caso, apenas um byte é alocado em memória, enquanto no segundo dois bytes são alocados devido ao caractere nulo de terminação do ***string***. Por exemplo, o caractere constante `'a'` ocupa apenas um byte, enquanto o ***string*** constante `"a"` ocupa dois bytes.

Outra tal diferença é que é normal atribuir um caractere constante ao conteúdo de um ponteiro para o tipo **char**, mas o mesmo não é verdade com relação 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 seguinte, apesar de legal, é desprovida de significado prático:

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

Como um ***string*** é interpretado como um ponteiro para **char** e um ponteiro referenciado tem o mesmo tipo do valor para o qual o ponteiro aponta, esta última instrução tenta atribuir o valor de um ponteiro (i.e., um endereço) a uma variável do tipo **char**. Por outro lado, é legal e normal atribuir um ***string*** a um ponteiro para o tipo **char**, mas, certamente, será problemático atribuir um caractere constante a um ponteiro. Por exemplo, se `p` é um ponteiro para **char**, a primeira das instruções a seguir é legal e perfeitamente apropriada, mas a segunda, apesar de legal, certamente irá trazer problemas quando o programa que contém tal instrução for executado.

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

É importante ainda chamar a atenção para uma confusão comum entre iniciantes em C: iniciações e atribuições não são eqüivalentes. Por exemplo, a iniciação:

```
char  *p = "string";
```

é legal e correta do ponto de vista prático, pois se está atribuindo um endereço a um ponteiro. Entretanto, a instrução:

```
*p = "string";
```

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

8.4 Funções de Biblioteca para Processamento de Strings

A biblioteca padrão da linguagem C possui várias funções para tratamento de ***strings***, que se encontram no módulo de biblioteca string. Portanto, para utilizá-las, use `#include <string.h>` em seu programa. Entrada e saída de ***strings*** utilizam funções do módulo de biblioteca `stdio`. Portanto, para utilizar estas funções, use `#include <stdio.h>`.

8.4.1 Entrada de Strings

Pode-se ler ***strings*** na entrada de dados padrão utilizando a função **scanf()** em conjunto com o especificador de formato **%s**. O argumento a ser utilizado com **scanf()** deve ser um array de caracteres com tamanho suficiente para conter o ***string*** introduzido. Após a leitura do ***string*** de entrada, **scanf()** automaticamente acrescenta o caractere nulo no final do ***string***. Por exemplo[1]:

```
char  str[30];
...
scanf("%s", str);
```

[1] O uso do operador **&** precedendo `str` não é necessário aqui, pois `str` já é um endereço, mas seu uso não causa erro num compilador ISO C.

Esta última instrução seria capaz de ler um ***string*** de no máximo 29 (e não 30) caracteres introduzidos pelo usuário do programa.

Um sério problema com a função **scanf()** usada com o especificador `%s` é que, por maior que seja o tamanho do array que se usa como argumento, não se pode garantir que seu tamanho será suficientemente grande para conter os caracteres introduzidos pelo usuário. Isto é, considerando novamente o exemplo anterior, quem garante que o usuário não irá introduzir mais de 29 caracteres? Portanto, o programador jamais deve usar simplesmente `%s` como especificador de formato de ***strings*** com a função **scanf()**. O uso deste especificador não apenas pode causar o mau funcionamento do programa, devido à corrupção de memória, como também torna o programa suscetível a ataques de ***hackers***[2]. Em ambos os casos, o problema reside no fato de a função **scanf()**, quando utilizada com o especificador `%s`, poder alterar porções de memória que não estão alocadas para o array utilizado como argumento.

Os problemas causados pelo uso do especificador **%s** em conjunto com **scanf()** podem ser facilmente resolvidos utilizando-se o seguinte especificador de formato:

%*n*s

onde *n* determina o número máximo de caracteres a ser lidos pela função **scanf()**. Assim, o modo seguro de chamar a função **scanf()** no último exemplo seria:

```
scanf("%29s", str);
```

Uma desvantagem do uso de **scanf()** para entrada de ***strings*** é que cada ***string*** termina quando o usuário digita um espaço em branco, o que significa que ***strings*** contendo caracteres em branco não podem ser introduzidos usando esta função. Uma forma aparente de superar esta deficiência é utilizar a função de entrada **gets()** que é específica para entrada de ***strings***.

---

[2] Discutir este problema de segurança está fora do escopo deste livro, mas o leitor não terá dificuldades em encontrar vastas discussões sobre o tema na Internet, especialmente em sites dedicados aos ***hackers***.

A função **gets()** recebe um único argumento que é um ponteiro para um array de caracteres. Caracteres são lidos do teclado até que o caractere `'\n'`, que representa `[ENTER]` ou `[RETURN]`, seja encontrado. Então, a função adiciona o caractere nulo e os caracteres obtidos são armazenados a partir do endereço recebido como argumento. Por exemplo:

```
char  str[30];
...
gets(str);
```

Como no caso da função **scanf()**, o array passado como argumento para **gets()** deve ter capacidade suficiente para conter os caracteres digitados pelo usuário. Entretanto, diferentemente do que ocorre com a função **scanf()**, a função **gets()** não possui meios para limitar o número de caracteres lidos no meio de entrada padrão. Portanto, para evitar os problemas enumerados anteriormente com o uso incorreto de **scanf()** em conjunto com o especificador `%s`, a função **gets()** jamais deve ser utilizada[3].

A função **fgets()**, apesar da semelhança com **gets()**, não padece do grave problema que acomete a função **gets()**. A função **fgets()** é descrita em detalhes no **Capítulo 12**. A **Seção 8.8** apresenta um método alternativo para a leitura de ***strings*** que tenta superar deficiências apresentadas pelas funções da biblioteca padrão de C utilizadas com este propósito.

8.4.2 Saída de Strings

Pode-se imprimir ***strings*** no meio de saída padrão utilizando a função **printf()** em conjunto com o especificador de formato `%s`. O argumento utilizado com **printf()** para impressão de ***strings*** no meio de saída deve ser um ponteiro para um array de caracteres contendo o caractere nulo. A função **printf()** imprime todos os caracteres no array até que o caractere nulo seja encontrado. Por exemplo:

---

[3] Você pode ficar intrigado com o fato de existir uma função na biblioteca padrão de C que se recomenda que nunca seja utilizada. Na realidade, a função **gets()** representa mais uma falha de design da linguagem C que se perpetua devido à compatibilidade histórica.

```
char  str[] = "isto e' um string"

printf("O string impresso e': %s.\n", str);
```

Esta última instrução causaria a impressão do seguinte no meio de saída padrão:

```
O string impresso e': isto e' um string.
```

Existe ainda a função **puts()** para saída de ***strings***, mas esta função não oferece nenhuma facilidade adicional que não seja oferecida pela função **printf()**. Por exemplo, a chamada de **puts()**:

```
puts(str);
```

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

8.4.3 Comprimento de Strings: Função strlen()

A função de processamento de ***strings*** mais simples é aquela que calcula o comprimento (i.e., o número de caracteres) de um ***string***. A função **strlen()** retorna o comprimento do ***string*** que recebe como argumento e tem o seguinte protótipo[4]:

```
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 argumento `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 7.5** pode ser relaxada, pois é raro ter-se 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.

[4] O tipo **size_t** é um tipo derivado definido na biblioteca padrão de C (v. **Seção 1.6.5**).

Para utilizar a função **strlen()**, deve-se incluir o arquivo de cabeçalho <string.h>[5]. A seguir, é apresentado um exemplo de uso desta função:

```
#include <string.h>

char    *ptr = "um string de comprimento 27";
size_t  j;

j = strlen(ptr); /* Resulta na atribuição de 27 a j */
```

Note que a função **strlen()** não inclui o caractere terminal de *string* '\0' na contagem do número de caracteres do *string* recebido como argumento.

Apesar de a função **strlen()** existir pronta para uso no módulo string da biblioteca padrão de C, é instrutivo examinar como esta função pode ser implementada em C. Provavelmente, se solicitado a escrever tal função, um programador iniciante em C procederia da seguinte maneira:

```
size_t  Comprimento1(const char *str)
{
   size_t tamanho = 0; /* Armazena o tamanho do string  */
   long   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;
}
```

[5] A função **strlen()**, bem como todas as funções de biblioteca apresentadas a seguir, requer a inclusão deste arquivo de cabeçalho ou o uso explícito dos protótipos destas funções. Claramente, é mais fácil incluir este arquivo de cabeçalho do que escrever cada um dos protótipos necessários.

O funcionamento da função `Comprimento1()` é simples e dispensa mais comentários além daqueles encontrados na própria função. Mas esta função pode ser melhorada notando que o valor da variável `i` assume os mesmos valores que a variável `tamanho`. Portanto, basta utilizar uma destas variáveis. Escolhendo a variável `tamanho` para tal, pode-se implementar uma nova versão para a função que calcula o comprimento de um ***string***, como:

```
size_t  Comprimento2(const char *str)
{
   size_t tamanho = 0; /* Armazena o tamanho do string e  */
                       /* é 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 `Comprimento2()` utiliza uma variável a menos que a função `Comprimento1()`. Uma outra novidade introduzida na função `Comprimento2()` é que a expressão:

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

utilizada como teste pela instrução **while** na função `Comprimento1()` foi substituída por simplesmente:

```
str[i]
```

Essas duas expressões são equivalentes porque, em qualquer código de caracteres, o valor de `'\0'` é 0.

As duas funções apresentadas acima funcionam perfeitamente bem, mas, provavelmente, um programador de C experiente escreveria tal função de modo mais elegante, como mostrado a seguir:

```
size_t  Comprimento3(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 `Comprimento3()` requer conhecimento da relação entre ponteiros e arrays unidimensionais e das propriedades de precedência e associatividade dos operadores ***** e **++**. Para compreender o funcionamento desta 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 a chave para compreensão desta função é a expressão:

```
*str++
```

Esta expressão envolve o uso de dois operadores, ***** e **++**, que fazem parte de um mesmo grupo de precedência (v. **Apêndice A**). Assim, para saber qual deles é aplicado primeiro, é necessário utilizar a propriedade de associatividade destes operadores, que, no caso de todos os operadores unários, é à direita. Portanto, 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 passagem pelo laço, o resultado desta 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 neste caractere. Se este 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 estará apontando para o segundo caractere do ***string*** e a história se repete, de modo que o laço encerrará quando o resultado desta expressão for o caractere nulo.

Em termos de funcionalidade ou eficiência, não há diferença entre as funções Comprimento2() e Comprimento3() apresentadas acima. Simplesmente, a função Comprimento3() é mais elegante, pois demonstra um conhecimento mais profundo da linguagem C por parte do programador.

8.4.4 Cópia de Strings: Função strcpy()

A função **strcpy**() recebe dois ***strings*** como argumentos e, após sua execução, o segundo ***string*** é copiado no primeiro. O protótipo desta função é:

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

Como exemplo de uso desta função, considere:

```
char *origem = "String a ser copiado";
char destino[50];

strcpy(destino, origem);
```

Ao final desta chamada, destino apontará para o ***string*** "String a ser copiado". Note que o primeiro argumento deve ter espaço suficiente para conter o ***string*** que será copiado.

A função **strcpy()** retorna o primeiro argumento (no último exemplo, esta função retornaria destino).

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

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

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

   return inicioStrDestino;
}
```

Se você entendeu o funcionamento da função `Comprimento3()` apresentada na seção anterior, certamente não terá dificuldades em entender a função `CopiaString()` apresentada aqui. Novamente, as expressões `*destino++` e `*origem++` referem-se, respectivamente, ao conteúdo das posições de memória para onde apontam os ponteiros `destino` e `origem`. Portanto, a expressão:

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

copia cada caractere apontado pelo ponteiro `origem` no conteúdo apontado por `destino` e o laço **while** termina quando for copiado o caractere nulo; i.e., quando o resultado da aplicação do operador de atribuição for zero[6].

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

É importante salientar que, quando a função **strcpy()** é chamada, o primeiro argumento deve ser um ponteiro para um array com capacidade suficiente para conter o resultado da operação e não simplesmente um ponteiro qualquer para o tipo **char**. Esta recomendação pode parecer

[6] 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 == pelo operador =, que é um engano que ocorre com freqüência. Aqui, todavia, não houve nenhum engano.

irrelevante para um programador experiente em C, mas refere-se a uma causa comum de erros de programação entre programadores inexperientes na linguagem.

8.4.5 Concatenação de Strings: Função strcat()

A função **strcat()** tem dois *strings* como argumentos e serve para concatenar uma cópia do segundo *string* ao final do primeiro. O caractere terminal do primeiro *string* é substituído pelo primeiro caractere do segundo *string*. O protótipo da função **strcat()** é :

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

Um exemplo de uso de **strcat()** é apresentado a seguir:

```
char  str[20] = "Bom ";
const char *ptr = "dia";

strcat(str, ptr);
```

Ao final dessa chamada de **strcat()**, o array `str` conterá o *string* `"Bom dia"`. Note (novamente) que o primeiro argumento deve ter espaço suficiente para conter o resultado da concatenação.

A função **strcat()** retorna o primeiro argumento (no último exemplo, esta função retornaria `str`).

Uma forma de concatenar *strings* constantes é simplesmente colocando-os justapostos, conforme foi apresentado na **Seção 1.2.3**. Neste caso, o número de espaços em branco entre os *strings* não faz diferença. Por exemplo, a instrução:

```
printf( "primeiro string..."  "segundo string..."
        "terceiro string" );
```

é tratada pelo compilador de C como se tivesse sido escrita desta maneira:

```
printf("primeiro    string...segundo    string...terceiro
string");
```

8.4.6 Casamento de Strings: Função strstr()

A função **strstr()** recebe dois ***strings*** como entrada e retorna um ponteiro para a posição da primeira ocorrência do segundo ***string*** no primeiro. Se nenhum **casamento** for encontrado, esta função retorna **NULL**. Esta função possui o seguinte protótipo:

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

Como exemplo de uso de **strstr()**, considere:

```
char *ptr1 = "bom dia";
char *ptr2 = "dia";
char *posicao;

posicao = strstr(ptr1, ptr2);
```

Após a chamada da função `strstr()` no exemplo acima, a variável `posicao` passa a apontar para a posição onde se encontra o caractere `'d'` no string `"bom dia"`.

8.4.7 Comparação de Strings: Função strcmp()

Dois ***strings*** são iguais quando eles têm o mesmo comprimento e contêm exatamente os mesmos caracteres nas respectivas posições. 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***. Um ***string*** é maior do que (i.e., sucede) outro se ele não é nem igual nem menor do que o outro. ***Comumente***, caracteres são comparados de acordo com a ordem determinada pelo valor inteiro atribuído a cada caractere no código de caracteres utilizado. Na maioria dos códigos de caracteres conhecidos, dígitos precedem letras maiúsculas, que, por sua vez, precedem letras minúsculas. As letras maiúsculas e minúsculas são ordenadas de acordo com a ordem alfabética usual. Exemplos: `"copo"` precede `"corpo"`, `"Carol"` precede `"Carolina"` e `"Maria"` precede `"maria"`.

É interessante notar que, quando se comparam ***strings*** consistindo em dígitos, alguns resultados podem surpreender. Por exemplo, o ***string***

"10" precede o ***string*** "8", apesar de o número 10 ser obviamente maior do o número 8. Isto ocorre porque os dois ***strings*** diferem nos primeiros caracteres correspondentes e '1' é menor do que '8'.

A função **strcmp()** é utilizada para comparar ***strings***. Ela recebe dois ***strings*** como entrada e retorna:

- 0, se os ***strings*** forem 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 protótipo da função **strcmp()** é :

```
int strcmp(const char *string1, const char *string2)
```

Considere os seguintes exemplos de uso da função **strcmp()**:

```
char ar1[20] = "Maria";
char ar2[10] = "Maria";
char *str1 = "maria";

strcmp(ar1, ar2); // Retorna 0 ("Maria" = "Maria")
strcmp(ar1, str1); // Retorna um valor negativo ("Maria" <
"maria")
strcmp(ar1, "MARIA"); // Retorna um valor positivo ("Maria" >
"MARIA")
```

Apesar da aparência, esta função não é muito útil quando se comparam ***strings*** numa linguagem contendo caracteres acentuados que possuem o mesmo valor em ordem alfabética que caracteres não acentuados, como ocorre com português. Por exemplo, a chamada da função:

```
strcmp("Mirtes", "Mércia");
```

resulta num valor menor do que zero, o que significa dizer que o ***string*** "Mirtes" é menor do que "Mércia", o que não corresponde ao esperado em português. Ou seja, em qualquer lista alfabética em português, o nome "Mércia" precede o nome "Mirtes", porque, em termos de ordenação em português, o caractere "é" tem o mesmo valor que o caractere "e". Acontece 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 estes valores. Além disso, quando o código de caracteres utilizado inclui caracteres acentuados[7], estes possuem valores inteiros associados maiores do que os caracteres não acentuados.

Concluindo, se seu problema é classificar *strings* que possam incluir caracteres acentuados de acordo com as regras de português, você não deve utilizar a função **strcmp()**. Nesta situação, utilize a função **strcoll()**, descrita em detalhes no **Volume II**. Esta função leva em consideração conceitos de localidade, que também são descritos no **Volume II**.

8.4.8 Procurando um Caractere num String: Funções strchr() e strrchr()

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, esta função retorna um ponteiro para a primeira ocorrência do caractere (segundo argumento) no *string* (primeiro argumento). Se o caractere não for encontrado, a função retorna **NULL**. Como exemplo de uso desta função, considere o seguinte programa:

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

int main()
{
   char string[80] = "Isto e' um teste.";

   printf("%s", strchr(string, 'e'));

   return 0;
}
```

Quanto executado, esse programa imprime:

```
e' um teste.
```

[7] O código ASCII original, por exemplo, não possui caracteres acentuados.

É importante notar que a função **strchr()** considera o caractere terminal de ***string*** `'\0'` como parte do ***string***. Assim, a chamada `strchr(str, '\0')` retorna um ponteiro para o caractere terminal do ***string*** `str`.

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

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

Quando bem-sucedida, esta função retorna um ponteiro para a última ocorrência do caractere (segundo argumento) no ***string*** (primeiro argumento). Se o caractere não for encontrado, a função retorna **NULL**. O seguinte programa apresenta um exemplo de uso da função **strrchr()**:

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

int main(void)
{
   char *str = "Apenas um string";
   char c = 's';

 printf("O restante de \"%s\" comecando com a ultima \n"
   "ocorrencia do caractere \'%c\' e' \"%s\"", str, c,
           strrchr(str, c));

   return 0;
}
```

Quanto executado, este programa imprime:

```
O restante de "Apenas um string" comecando com a ultima ocorrencia
do caractere 's' e' "string"
```

A função **strrchr()** também considera o caractere terminal de ***string*** como parte do ***string***, de modo que a chamada `strrchr(str, '\0')` tem o mesmo efeito que a chamada `strchr(str, '\0')`.

8.4.9 Procurando Caracteres num String: Funções strpbrk(), strspn() e strcspn()

A função **strpbrk()** procura em um ***string*** a primeira ocorrência de qualquer caractere presente em outro ***string***, e seu protótipo é:

```
char *strpbrk(const char *str1, const char *str2)
```

Onde:

- `str1` é o ***string*** no qual será efetuada a busca
- `str2` é o ***string*** que contém os caracteres que serão procurados em `str1`

Portanto, a função **strpbrk()** retorna um ponteiro para o primeiro caractere de `str2` encontrado em `str1`. Se nenhum caractere de `str2` for encontrado em `str1`, esta função retorna **NULL**. Como exemplo de uso desta função, considere o seguinte programa:

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

int main(void)
{
   char *str1 = "Apenas um string";
   char *str2 = "Teste";
   char *ocorrenciaPtr;

   ocorrenciaPtr = strpbrk(str1, str2);

   if (ocorrenciaPtr)
      printf("Dentre os caracteres em \"%s\", \'%c\' "
             "e' o primeiro a aparecer em \"%s\"\n",
             str2, *ocorrenciaPtr, str1);
   else
       printf("Nao existe caractere em \"%s\" que apareca em \
"%s\"\n",
             str2, str1);

   return 0;
}
```

Quando executado, o programa acima produz como resultado:

```
Dentre os caracteres em "Teste", 'e' e' o primeiro a aparecer em
"Apenas um string"
```

A função **strspn()**, cujo protótipo é apresentado a seguir, é, em alguns aspectos, semelhante à função **strpbrk()** vista acima:

```
size_t  strspn(const char *str1, const char *str2)
```

A função **strspn()** retorna o índice do primeiro caractere do ***string*** `str1` que não se encontra em `str2`[8].

A função **strcspn()**, cujo protótipo é apresentado a seguir, é semelhante à função **strspn()**:

```
size_t  strcspn(const char *str1, const char *str2)
```

A função **strcspn()** retorna o índice do primeiro caractere do ***string*** `str1` que é igual a qualquer caractere no ***string*** `str2`.

Como exemplo de uso das funções **strspn()** e **strcspn()**, considere o seguinte programa:

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

int main(void)
{
   char    *str1 = "ABCDEFGHIJH";
   char    *str2 = "DCFAB";
   size_t  posicao;

   posicao = strspn(str1, str2);

   printf( "Primeiro caractere de \"%s\" que nao "
           "esta em \"%s\": %c\n\n",
            str1, str2, str1[posicao]);

   posicao = strcspn(str1, str2);
```

[8] Alternativamente, pode-se dizer que esta função retorna o comprimento do segmento inicial contido de `str1` que contém todos os caracteres de `str2`. Mas a descrição apresentada acima parece ser mais clara.

```
   printf( "Os strings \"%s\" e \"%s\" se "
           "interceptam no caractere %c\n",
           str1, str2, str1[posicao]);

   return 0;
}
```

A execução do programa acima produz como resultado a impressão:

```
Primeiro caractere de "ABCDEFGHIJH" que nao esta em "DCFAB": E
Os strings "ABCDEFGHIJH" e "DCFAB" se interceptam no caractere A
```

Tanto a função **strspn()** quanto a função **strcspn()** levam o caractere terminal de ***string*** em consideração.

8.4.10 Separando Strings em Tokens: Função strtok()

Um **token** é uma seqüência de caracteres considerada como uma unidade que possui significado próprio num determinado contexto. ***Tokens*** num string são identificados pelos **separadores** que os cercam. Por exemplo, as partes (***tokens***) que compõem um comando (***string***) do sistema operacional Unix são separadas por espaços em branco ou outros separadores permitidos por este sistema.

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

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

onde:

- `string` é o ***string*** a ser dividido em partes (***tokens***)
- `separadores` é um ***string*** contendo os separadores das partes

A primeira chamada de **strtok()** retorna um ponteiro para o primeiro caractere do primeiro ***token*** encontrado em `string` e um caractere terminal de ***string*** é colocado neste parâmetro ao final do ***token*** retornado. Chamadas subseqüentes da função com **NULL** como primeiro argumento irão retornar os ***tokens*** seguintes até que nenhum deles seja remanescente.

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 separador de *tokens* seja espaço em branco, a **Figura 27** ilustra três chamada consecutivas da função **strtok()**; na primeira delas, o string "Um Dois Tres" é passado como primeiro argumento da função, enquanto nas demais chamadas o primeiro argumento é **NULL**. A **Figura 27** mostra, para cada chamada da função **strtok()**, o ponteiro retornado e o conteúdo do *string* original alterado pela função.

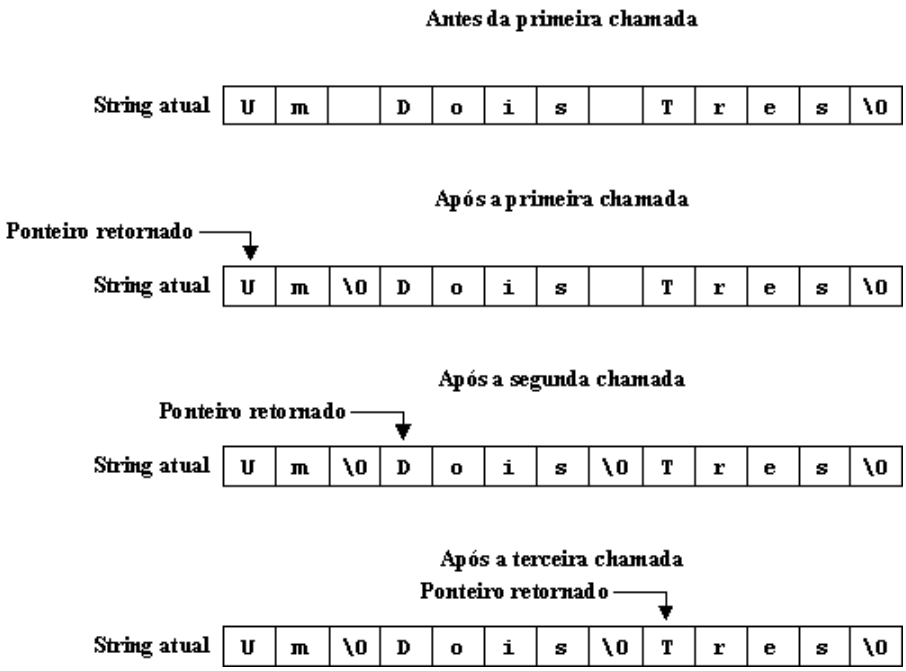

Figura 27: Funcionamento da função strtok()

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

   token = strtok(string, separadores);

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

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

   return 0;
}
```

Quando executado, o programa imprime o seguinte no meio de saída padrão:

```
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 salientar que a função **strtok()** modifica o *string* passado como argumento, conforme mostra a **Figura 27**. Portanto, se necessário, faça uma cópia do mesmo antes de chamar a função. Deve-se notar, ainda, que o *string* contendo os separadores (i.e., o segundo argumento da função) pode ser diferente em duas chamadas sucessivas. Finalmente, não utilize a função **strtok()** como argumento de uma função que também utiliza **strtok()**, pois isto poderá acarretar um laço infinito.

8.4.11 Outras Funções para Processamento de Strings

Além das funções para processamento de *strings* descritas aqui, existem muitas outras funções com esta e outras finalidades no módulo string da biblioteca padrão de C. Estas funções são resumidas na **Tabela 30,** a seguir.

FUNÇÃO	O QUE FAZ
memchr()	Procura a primeira ocorrência de um caractere num *string*, limitando a busca aos n primeiros caracteres.
memcmp()	Compara n bytes de dois blocos de memória.
memcpy()	Copia n bytes de um bloco de memória para outro.
memmove()	Copia n bytes de um bloco de memória para outro; os blocos de origem e de destino podem se sobrepor.
memset()	Preenche n bytes de um bloco de memória com um dado valor.
strcoll()	Compara dois *strings* usando a seqüência de ordenação da localidade corrente selecionada por **setlocale()** e retorna um valor indicando se eles são iguais ou se um deles é menor do que o outro.
strerror()	Retorna um *string* descrevendo o erro associado com o valor recebido como argumento.
strncat()	Concatena, no máximo, n caracteres de um *string* no final de outro string.
strncmp()	Compara no máximo n caracteres de dois *strings* e retorna um valor indicando se eles são iguais ou se um deles é menor do que o outro.

strncpy()	Copia, no máximo, n caracteres de um ***string*** para outro. Se n for maior do que ou igual ao comprimento do ***string*** de origem, apenas n caracteres serão copiados; caso contrário, a função copia os n primeiros caracteres do ***string*** de origem para o ***string*** de destino e acrescenta o caractere `'?'` ao ***string*** de destino.
strxfrm()	Copia um ***string***, transformado usando-se a seqüência de ordenação **LC_COLLATE** estabelecida por **setlocale()** para outro ***string***. Pode-se limitar o número de caracteres que serão copiados.

Tabela 30: Outras funções para processamento de strings

Observe, na **Tabela 30**, que algumas funções do módulo string, como **memcpy()** e **memmove()**, manipulam bytes e não strings como a maioria das funções do módulo. Maiores detalhes sobre como utilizar cada uma das funções apresentadas na **Tabela 30** podem ser encontrados no **Volume II**.

8.5 A Função main()

A função **main()**, que já foi apresentada informalmente, é uma função com certas características especiais. A presença desta função num programa em C é obrigatória em **programas hospedados,** i.e., programas que são executados sob intermediação de um sistema operacional. Programas que não são hospedados, denominados **programas embutidos** ou **embarcados**, não precisam atender a esta exigência[9]. Esta função é sempre a primeira função a ser executada no programa e, quando ocorre o retorno dela, o programa é encerrado.

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

```
int main(void)
```

[9] Este livro trata apenas de programas hospedados. Portanto, aqui, ***programa*** significa ***programa hospedado***.

Entretanto, outro fato interessante sobre a função **main()** é que ela pode receber dois argumentos do sistema operacional no qual o programa é executado. Estes argumentos estão associados a valores que acompanham a chamada do programa e são denominados **parâmetros de linha de comando**. A definição de uma função **main()** contendo estes dois argumentos tem o seguinte protótipo:

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

O primeiro argumento recebido pela função **main()** quando o programa executado é usualmente denominado `argc` e representa o número de parâmetros presentes na linha de comando do sistema operacional quando o programa é invocado. O segundo argumento fornecido pelo sistema operacional, usualmente denominado `argv`, consiste em um array de ***strings*** que representam os argumentos presentes na linha de comando quando o programa é invocado.

Para que parâmetros possam ser passados para o programa, eles devem seguir o nome do programa na linha de comando, de acordo com o seguinte formato[10]:

nome-do-programa parâmetro1 parâmetro2 ... parâmetroN

Cada ***string*** na chamada do programa deve ser separado do outro por meio de um ou mais espaços em branco. Se um dado parâmetro (***string***) possui espaço em branco em seu interior, ele deve ser colocado entre aspas.

O nome do programa é armazenado como primeiro item no array `argv` e, em seguida, os outros ***strings*** presentes na linha de comando serão armazenados neste array. O valor de `argc` será automaticamente atribuído como sendo o

[10] Na verdade, o modo como parâmetros são passados para um programa depende do sistema operacional utilizado e, portanto, não é especificado pelo padrão de C. O formato apresentado aqui é o mais comumente utilizado. Os sistemas operacionais DOS e aqueles da família Unix utilizam este formato. Se você estiver depurando um programa que requeira argumentos no gdb, utilize o comando `run`, conforme visto na **Seção 6.13**, e acrescente, em seguida, os argumentos do programa do mesmo modo que eles seriam escritos na linha de comando.

número de elementos no array `argv`. Por exemplo, se houver três parâmetros na linha de comando, `argv` terá quatro elementos (os três parâmetros mais o nome do programa) e `argc` assumirá automaticamente o valor 4.

Como exemplo mais concreto, 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]);
}
```

Quando executado, o programa acima apresenta no meio de saída padrão o valor de `argc` e os ***strings*** armazenados em `argv`. Por exemplo, suponha que o nome deste programa após compilado seja `exemplo`. Então, como resultado da execução deste programa no Linux usando o seguinte na linha de comando:

```
./exemplo  azul preto branco
```

o programa produziria como saída[11]:

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

A função **main()** de C sempre teve **int** como tipo de retorno. Entretanto, por diversas razões, o valor retornado muitas vezes é desprezado pelo programador. Inclusive, alguns compiladores incluem extensões que permitem que esta função seja definida com tipo de retorno **void**.

[11] O modo como o nome do programa é passado para o programa depende do sistema operacional. Aqui, o sistema operacional inclui o caminho relativo até o programa.

No padrão C99, a função **main()** continua tendo tipo **int**, mas compiladores que seguem este padrão acrescentam a instrução `return 0;` em funções **main()** que não incluem explicitamente instruções de retorno. Isto ocorre não apenas na ausência completa de instrução de retorno na função **main()**, mas também quando existe uma tal instrução que só é executada condicionalmente. Por exemplo, a função **main()**:

```
int main(void)
{
   if (...)
      return 1;
   ...
}
```

teria automaticamente acrescentada a instrução `return 0;` logo antes do final desta função, a não ser que haja uma outra instrução de retorno fora do corpo de uma instrução condicional.

8.6 O Identificador Predefinido __func__ (C99)

O identificador predefinido **__func__**[12], que foi introduzido pelo padrão C99, pode ser utilizado no corpo de uma função, e, quando isto ocorre, o compilador acrescenta a seguinte definição logo no início do corpo da função:

```
static const char __func__[] = "nome da função";
```

Assim, por exemplo, a seguinte chamada de **printf()** imprimiria o nome da função onde esta chamada se encontra:

```
void UmaFuncao()
{
   ...
   printf("%s\n", __func__);
   ...
}
```

[12] Note que o nome deste identificador começa e termina com dois caracteres de sublinha.

No exemplo anterior, o compilador acrescentaria a definição de **__func__**, de tal modo que a definição da função ficaria assim:

```
void UmaFuncao()
{
   static const char __func__[] = "UmaFuncao";
   ...
   printf("%s\n", __func__);
   ...
}
```

O identificador predefinido **__func__** é utilizado tipicamente da mesma maneira que as macros predefinidas **__LINE__** e **__FILE__** (v. **Seção 5.3.8**), mas este identificador, apesar da aparência, não representa uma macro. Ele não poderia ser uma macro simplesmente porque o pré-processador não tem conhecimento sobre funções.

8.7 Caracteres Universais (C99)

Caracte res universais (**UCN**) são caracteres que constam no código de caracteres especificado pelo padrão ISO/IEC 10646, que engloba praticamente todos caracteres de todas as linguagens naturais conhecidas[13]. De acordo com o padrão C99, caracteres universais podem fazer parte de identificadores, caracteres constantes ou ***strings*** constantes. Estes caracteres são utilizados para designar caracteres que não estão no código de caracteres básico de um compilador.

Existem duas maneiras de especificar caracteres universais; ambas utilizam a seqüência de escape `\u` (ou `\U`) seguida de um número com quatro ou oito dígitos na base hexadecimal que corresponde ao valor associado a um caractere no código de caracteres especificado pelo padrão ISO/IEC 10646. Estes modos de especificação de caracteres universais são apresentados na **Tabela 31**.

[13] Existem cerca de cem mil caracteres no código de caracteres definido pelo padrão ISO/IEC 10646, que é semelhante em vários aspectos ao código Unicode. Por exemplo, os caracteres e suas respectivas codificações são os mesmos em ambos os padrões.

FORMATO DE CARACTER UNIVERSAL	CODIFICAÇÃO ISO/IEC 10646
\uNNNNNNNN	NNNNNNNN
\uNNNN	0000NNNN

Tabela 31: Formatos de caracteres universais

Conforme foi afirmado, de acordo com C99, identificadores podem ser construídos utilizando caracteres locais a uma determinada língua natural. No entanto, nem todo caractere utilizado numa linguagem natural pode ser utilizado na formação de um identificador. O Anexo D do padrão C99 especifica precisamente quais caracteres podem ser utilizados em identificadores. Como regra prática, assuma que qualquer caractere considerado letra numa determinada língua natural pode ser utilizado em identificadores. Outros caracteres (como, por exemplo, `'ª'`) podem também ser utilizados em identificadores, mas provavelmente você deve estar interessado apenas em utilizar letras da língua portuguesa que, em padrões ISO anteriores a C99, não eram permitidas em identificadores (por exemplo, ç, ã, é)[14].

Um caractere universal não pode especificar um valor no intervalo de `D800` até `DFFF` nem um valor menor do que `00A0`, a não ser `0024` (`$`), `0040` (`@`), ou `0060` (`` ` ``).

8.8 Leitura Segura de Strings

As funções da biblioteca padrão de C que podem ser usadas para leitura de ***strings*** não são suficientemente adequadas para esta finalidade básica. Nesta seção, serão descritos alguns problemas apresentados pelas funções da biblioteca padrão de C utilizadas na leitura de ***strings***. Em seguida, será proposta uma função com esta finalidade que supera os problemas apontados. Finalmente, será apresentado um programa que demonstra o uso da função que será especificada e implementada aqui.

[14] Não se anime muito com esta nova característica, pois até o momento de término da escrita deste livro, não havia nenhum compilador que permitisse, por exemplo, o uso de um identificador denominado `Ação`.

8.8.1 Problemas com as Funções de Leitura de Strings da Biblioteca Padrão

Serão apresentados a seguir alguns problemas que tornam as funções da biblioteca padrão de C freqüentemente utilizadas na leitura de ***strings*** inadequadas para esta finalidade.

► Função scanf()

O problema mais sério que ocorre com a função **scanf()** é que ela simplesmente não é capaz de ler ***strings*** contendo espaços em branco.

► Função gets()

A função **gets()** é provavelmente a função mais criticada de todas as funções da biblioteca padrão de C e aconselha-se radicalmente nunca utilizá-la. O problema mais grave com **gets()** é que ela pode corromper memória quando o usuário digita mais caracteres do que o array passado como argumento pode comportar. Lembre-se de que não se deve fazer suposições otimistas sobre o nível cognitivo dos usuários ou suas intenções.

► Função fgets()

A função **fgets()** é bem mais segura do que **gets()**, já que se pode especificar o número máximo de caracteres que podem ser lidos no meio de entrada. O problema com a função **fgets()** é que ela inclui o caractere `'\n'` (que representa o pressionamento da tecla `[ENTER]` ou `[RETURN]`) e, muito provavelmente, quando se solicita a introdução de um ***string*** no meio de entrada padrão, não se está interessado em processar este caractere.

8.8.2 Algoritmo para Leitura de Strings

A função de leitura de ***strings***, cuja implementação será apresentada em seguida, possui dois argumentos:

- `string`: este argumento de saída é um array de caracteres que receberá o ***string*** lido.
- `nElementos`: argumento de entrada que informa o tamanho (i.e., número de elementos) do array `string`.

A função de leitura segue o seguinte algoritmo:

1. Enquanto não faltar apenas um caractere para completar o array `string` ou um caractere de quebra de linha `'\n'` não for encontrado, faça o seguinte:

   1.1 Leia um caractere do meio de entrada padrão

   1.2 Se o caractere não for um caractere de quebra de linha, coloque-o no array (`string`)

2. Coloque o caractere terminal de ***string*** `'\0'` no array `string`

3. Verifique se o ***buffer*** de entrada foi esvaziado:

   3.1 Se o ***buffer*** de entrada estiver vazio ou contiver apenas o caractere `'\n'`, retorne um ponteiro para o ***string*** lido.

   3.2 Se o ***buffer*** de entrada não estiver vazio e contiver caracteres além de `'\n'`, retorne o ponteiro nulo indicando que o usuário digitou um número de caracteres maior do que o array podia suportar.

Note que uma das condições de parada do laço do passo 1 é faltar um caractere para completar o array, e não o fato de este array estar completo. Isto ocorre porque é necessário deixar espaço para completar o array com o caractere terminal de ***string*** `'\0'`.

8.8.3 Implementação da Função de Leitura de Strings

A função `LeString()` apresentada a seguir tenta ler ***strings*** no meio de entrada padrão usando **getchar()** e retorna o ***string*** lido apenas quando o número de caracteres lidos estiver dentro do número máximo de caracteres permitido. (As linhas de interesse da função foram numeradas para facilitar a discussão que segue a apresentação da função.)

```
char *LeString(char * string, unsigned nElementos)
{
   unsigned contador = 0; /* Conta o número de caracteres
lidos */
   char     caractereLido;
1.
   caractereLido = getchar();
2.   while ((caractereLido != '\n') && (contador < nElementos-
1)) {
3.       string[contador++] = caractereLido;
4.       caractereLido = getchar();
   }
5.
   string[contador] = '\0'; /* Termina o string */
6.   if ((caractereLido == '\n') || (getchar() == '\n')) //
Tudo bem
7.       return string;
8.   else {      /* Usuário digitou caracteres demais */
9.       LimpaBuffer();
         return (char *) 0;
   }
}
```

► **Comentários sobre a função LeString()**

1. Esta instrução simplesmente lê o primeiro caractere.

2. Esse laço **while** será responsável pela leitura dos caracteres restantes e o subseqüente armazenamento dos mesmos no array `string`. Existem duas condições de parada deste laço: (1) ter encontrado o caractere `'\n'` ou (2) ter lido no máximo `nElementos - 1` caracteres.

3. Aqui, é feito o armazenamento do caractere lido no array `string` e, ao mesmo tempo, incrementa-se a variável `contador`. É essencial que este incremento seja sufixo.

4. Simplesmente, lê o próximo caractere.

5. Neste ponto, a variável `contador` contém o índice do caractere que segue o último caractere armazenado no array `string`. Isto ocorre porque o operador sufixo de incremento foi utilizado. Esta instrução coloca o caractere terminal de ***string*** `'\0'` na posição indicada pela variável `contador`.

6. A instrução **if** testa o último caractere lido (armazenado na variável `caractereLido`) e o próximo caractere [obtido com **getchar()**] com `'\n'`.

7. Se o último caractere lido for `'\n'`, o usuário digitou menos caracteres do que o máximo permitido. Se o próximo caractere no ***buffer*** de entrada for `'\n'`, o usuário digitou exatamente o máximo permitido. Nestes dois casos, tudo ocorreu normalmente e resta apenas retornar o endereço do array `string`.

8. Se nem o último caractere lido nem o próximo caractere no ***buffer*** de entrada forem `'\n'`, o usuário digitou caracteres além do permitido.

9. Neste caso, limpa-se o ***buffer*** de entrada e retorna-se um ponteiro nulo.

8.8.4 Limpeza do Buffer de Entrada

A função `LimpaBuffer()`, apresentada a seguir, faz a faxina no buffer de entrada padrão. Isto é, a função `LimpaBuffer()` lê e descarta todos os caracteres que porventura tenham sido deixados no buffer de entrada em alguma tentativa de leitura de dados. Uma limitação desta função é que, se não houver pelo menos um caractere `'\n'` no buffer de entrada, ela irá aguardar a introdução deste caractere antes de liberar a entrada.

```
void LimpaBuffer(void)
{
    int valorLido = getchar(); /* valorLido deve ser int! */

        /* Lê caracteres no buffer de entrada enquanto */
        /* não encontra final de linha ou final de arquivo */
    while ((valorLido != '\n') && (valorLido != EOF))
        valorLido = getchar();}
}
```

► **Comentários sobre a função LimpaBuffer()**

A utilidade desta função já foi suficientemente discutida nas **Seções 2.6.1** e **3.7.4**. Conforme você já deve ter percebido, esta função pode ser implementada de modos diferentes.

A variável local `valorLido`, que armazena os caracteres deixados no *buffer*, deve ser do tipo **int** (e não do tipo **char**) porque existe a possibilidade de **getchar()** retornar um valor (**EOF**) que indica tentativa de leitura além do final de arquivo, e este valor é do tipo **int**. O laço **while** simplesmente lê todos os caracteres encontrados no buffer de entrada padrão e pára quando encontra o caractere `'\n'` (representando `[ENTER]` digitado pelo usuário) ou quando **getchar()** retorna **EOF**[15].

8.8.5 Uso da Função de Leitura

A função **main()** a seguir aceita qualquer *string* que respeite o número máximo de caracteres permitido e encerra apenas quando um *string* dentro desse limite for lido[16].

---

[15] No caso de entrada de dados padrão, o final de arquivo é simulado por meio da digitação da tecla `[CTRL]` seguida de `[D]`, no DOS, e da digitação da tecla `[CTRL]` seguida de `[Z]` no Unix.

[16] As linhas de interesse da função foram numeradas para facilitar a discussão que segue a apresentação da função.

```
int main(void)
{
1.    char strEntrada[TAMANHO_STRING];
2.    char *stringLido;

      printf("\nIntroduza um string com no maximo %d
caracteres: ",
           TAMANHO_STRING - 1);
3.    stringLido = LeString(strEntrada, TAMANHO_STRING);

4.    while(1) {
5.       if (stringLido) {
            printf("\nO string lido foi: %s\n", stringLido);
            break;    /* Tchau while */
         }
6.          printf("\nO string introduzido tem mais de %d
caracteres: ",
               TAMANHO_STRING - 1);
     printf("\nDigite um string com no maximo %d caracteres:
",
               TAMANHO_STRING - 1);
       stringLido = LeString(strEntrada, TAMANHO_STRING);
   }
   return 0;
}
```

▶ Comentários sobre a função main()

1. O array `strEntrada` armazena os ***strings*** lidos e `TAMANHO_STRING` é uma macro (constante simbólica) definida no arquivo que contém a função **main()**. Esta constante representa o número de elementos do array `strEntrada`, que irá conter o ***string*** introduzido pelo usuário.

2. O ponteiro `stringLido` apontará para o ***string*** lido se este for válido ou conterá um valor nulo em caso contrário.

3. Esta instrução tenta ler um ***string*** usando a função `LeString()`.

4. Esse laço termina com uma instrução **break** em seu corpo, que será executada apenas quando um ***string*** válido for lido.

5. Esta instrução **if** testa se o ***string*** foi lido sem problemas, isto é, se a variável `stringLido` contém um valor diferente de **NULL**. Se este for o caso, imprime-se o ***string*** lido e sai-se do laço.

6. Se a instrução **break** não foi executada, o ***string*** não foi lido corretamente (i.e., a variável `stringLido` é nula). As instruções seguintes fazem uma nova tentativa de leitura.

8.9 Exercícios de Revisão

1. Um aluno de programação escreveu a seguinte função com o objetivo de concatenar dois ***strings***. Descubra os erros de programação apresentados por esta função.

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

char *Concatena(char *str, const char *ptr)
{
   while (*str++)
      str++;
   while (*str++ = *ptr++)
      ; /* VAZIA */
   return str;
}
```

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

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

3. O seguinte programa aparece como exemplo num famoso livro programação em C. O que há de gravemente errado neste programa?

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

void main(void)
{
   int p;
   p = strcat(s1, s2);
}
```

4. A função `CopiaString()` apresentada a seguir 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 no string destino       */
   while (*origem++ = *++destino)
      ; /* Não há mais nada a fazer */

   return inicioStrDestino;
}
```

5. Compare a função `LeString()` apresentada na **Seção 8.8** com as funções da biblioteca padrão de C utilizadas em leitura de ***strings*** descritas na **Seção 8.4.1**.

6. O que ocorreria se o operador de incremento prefixo tivesse sido utilizado na linha 3 da função `LeString()` apresentada na **Seção 8.8**?

8.10 Exercícios de Programação

EP8.1) Escreva uma função ***recursiva*** que calcule o comprimento de um ***string***. Compare esta função com aquelas apresentadas na **Seção 8.4.3** em termos de eficiência (espaço ocupado em memória, rapidez) e legibilidade.

EP8.2) Escreva uma função, denominada `TransformaStr()`, que receba um ***string*** como primeiro argumento e um caractere como segundo

argumento. Quando o segundo argumento for o caractere `'M'`, esta função deve transformar o ***string*** de tal modo que todas as letras do ***string*** sejam maiúsculas. Quando o segundo argumento for o caractere 'm', esta função deve transformar o ***string*** de tal modo que todas as letras do ***string*** sejam minúsculas. Quando o segundo argumento não for `'M'` ou `'m'`, esta função não deve promover nenhuma transformação no ***string***. A função deverá retornar um ponteiro para o string transformado. (**Sugestão**: Utilize as funções **tolower()** e **toupper()** do módulo ctype da biblioteca padrão de C.)

EP8.3) Implemente uma função, denominada `Concatena()`, funcionalmente equivalente à função **strcat()**.

EP8.4) Implemente uma função, denominada `PosicaoEmString()`, funcionalmente equivalente à função **strcat()**.

EP8.5) Implemente uma função, denominada `ComparaStrings()`, funcionalmente equivalente à função **strcmp()**.

EP8.6) Implemente uma função, denominada `EncontraPrimeiroChar()`, funcionalmente equivalente à função **strchr()**.

EP8.7) Implemente uma função, denominada `EncontraUltimo Char()`, funcionalmente equivalente à função **strrchr()**.

EP8.8) Escreva uma função semelhante à função **strcmp()** que compare ***strings*** sem levar em consideração se as letras que compõem os ***strings*** são minúsculas ou maiúsculas. (**Sugestão**: Utilize a função **tolower()** do módulo ctype da biblioteca padrão de C.)

EP8.9) Escreva uma função que retorne um ponteiro para o caractere terminal de um ***string***. Você seria capaz de escrever esta função utilizando apenas uma instrução?

EP8.10) Escreva uma função que insira um ***string*** no interior de outro e retorne um ponteiro para o ***string*** alterado. O protótipo da função deve ser:

```
char *InsereStr(char *str, const char *strAInserir,
                unsigned posicao)
```

EP8.11) Escreva uma função que retorne 1 quando um ***string*** possuir apenas letras e dígitos ou 0 em caso contrário. (**Sugestão**: Utilize a função **isalnum()** do módulo ctype da biblioteca padrão de C.)

EP8.12) Escreva uma função que substitua todas as ocorrências num ***string*** de um dado caractere por outro caractere.

EP8.13) Escreva uma função que remova todas as ocorrências de um dado caractere num ***string***. Por exemplo, quando o ***string*** for `"casa"` e o caractere a ser removido for `'a'`, o ***string*** resultante será `"cs"`.

EP8.14) Escreva uma função que ***centralize*** um ***string*** no array que o contém. Isto é, os caracteres que compõem o ***string*** devem ocupar as posições centrais do array, as demais posições do array devem conter espaços em branco (exceto a última posição que deve conter o caractere `'\0'`).

EP8.15) Escreva uma função que ***alinhe à direita*** um ***string*** no array que o contém. Isto é, os caracteres que compõem o ***string*** devem ocupar as posições finais do array (exceto a última posição, que deve conter o caractere `'\0'`) e as posições iniciais do array devem conter espaços em branco.

EP8.16) Escreva uma função que concatene um número não especificado de ***strings*** ao final do string representado pelo primeiro argumento. Esta função deve possuir uma lista de argumentos variáveis (v. **Seção 3.4**) e retornar um ponteiro para o ***string*** que recebe o resultado da concatenação.

EP8.17) Escreva uma função que substitua caracteres de tabulação por espaços em branco.

EP8.18) Escreva uma função que remove todos caracteres não-gráficos de um ***string***. (**Sugestão**: Use a função **isgraph()** do módulo ctype da biblioteca padrão de C.)

EP8.19)

(a) Escreva uma função que retorne o número de ocorrências de um caractere num ***string***. O protótipo da função deve ser:

```
unsigned ContaOcorrencias(const char * oString,
                          char oCaractere)
```

(b) Escreva um programa que receba um ***string*** e um caractere do usuário e use esta função para determinar quantas vezes o caractere introduzido ocorre no ***string***. Este programa deve encerrar quando o usuário digitar `[RETURN]` como ***string***. (**Sugestão**: Utilize a função `LeString()` apresentada na **Seção 8.8** para ler os ***strings*** introduzidos pelo usuário.)

Como exemplo de interação deste programa, considere:

```
[Apresentação do programa]
Introduza uma cadeia de caracteres (maximo = 20 caracteres):
banana
Introduza um caractere: n
Numero de ocorrencias: 2
Introduza uma cadeia de caracteres (maximo = 20 caracteres):
[ENTER]
[Despedida graciosa do programa]
```

Neste exercício, não utilize ponteiros para acessar os caracteres dos ***strings*** (i.e., utilize apenas índices para esta finalidade).

EP8.20)

(a) Escreva uma função que encontre todas as ocorrências de um ***string*** em outro e as substitua por um outro ***string***. Esta operação deve ocorrer apenas quando o ***string*** a ser substituído for do mesmo tamanho do novo ***string***. A função deve retornar

o ***string*** substituído se a operação for bem-sucedida ou **NULL** em caso contrário. Esta função ainda deve contar o número de substituições efetuadas. O protótipo da função deve ser:

```
char *SubstituiOcorrencias( char *oString,
                            unsigned *numSubstituicoes,
                            const char *antigo,
                            const char *novo, )
```

(b) Escreva um programa que receba três ***strings*** do usuário e use esta função para substituir todas as ocorrências do segundo ***string*** no primeiro ***string*** pelo terceiro ***string***. O programa deve terminar quando o usuário digitar `[ENTER]` como primeiro ***string***. Exemplo de interação do programa:

```
[Apresentação do programa]
Introduza uma cadeia de caracteres: banana
Cadeia de caracteres que sera' substituida: na
Cadeia de caracteres que substituira': ta
A cadeia de caracteres original foi: 'banana'
O string modificado e': 'batata'
Numero de substituicoes: 2

Introduza uma cadeia de caracteres: bola
Introduza a cadeia de caracteres que sera' substituida: ola
Introduza a cadeia de caracteres que ira' substituir: oi
Erro: O string de substiuicao e o que sera' substituido devem
ter o mesmo tamanho

Introduza uma cadeia de caracteres: bola
Introduza a cadeia de caracteres que sera' substituida: ta
Introduza a cadeia de caracteres que ira' substituir: oi
O string original foi 'bola'
Nao foram encontradas ocorrencias de 'ta' para substituir
por 'oi'

Introduza uma cadeia de caracteres: [ENTER]
[Despedida graciosa do programa]
```

Neste exercício, não utilize ponteiros para acessar os caracteres dos ***strings*** (i.e., utilize apenas índices para esta finalidade).

Sugestão

Você pode utilizar a função **strstr()** para encontrar as ocorrências do ***string*** antigo no ***string*** `oString`, mas lembre-se de que esta função retorna um ponteiro apenas para a primeira ocorrência encontrada.

EP8.21)

(a) Reescreva as funções dos exercícios **EP8.19** e **EP8.20** de modo que estas funções utilizem apenas ponteiros para acessar os caracteres dos ***strings***.

(b) Escreva um programa que ofereça as duas opções de operações implementadas por estas funções, além da opção de sair do programa, e execute a operação correspondente à escolha do usuário. Considere o seguinte exemplo de interação:

```
[Apresentação do programa]
Escolha uma das opcoes a seguir:
1 - Conta o numero de ocorrencias de um caractere num
string
2 - Substitui ocorrencias de um string por outro
3 - Sai do programa
Sua escolha: 1

Introduza uma cadeia de caracteres: banana
Cadeia de caracteres que sera' substituida: na
Cadeia de caracteres que substituira': ta
A cadeia de caracteres original foi: 'banana'
O string modificado e': 'batata'
Numero de substituicoes: 2

Digite um caractere para prosseguir
[Limpeza de tela]
Escolha uma das opcoes a seguir:
1 - Conta o numero de ocorrencias de um caractere num
string
2 - Substitui ocorrencias de um string por outro
3 - Sai do programa
Sua escolha: 2

Introduza uma cadeia de caracteres: banana
Cadeia de caracteres que sera' substituida: na
```

```
Cadeia de caracteres que substituira': ta
A cadeia de caracteres original foi: 'banana'
O string modificado e': 'batata'
Numero de substituicoes: 2

Digite um caractere para prosseguir
[Limpeza de tela]
Escolha uma das opcoes a seguir:
1 - Conta o numero de ocorrencias de um caractere num string
2 - Substitui ocorrencias de um string por outro
3 - Sai do programa
Sua escolha: 3
[Despedida graciosa do programa]
```

EP8.22) Escreva um programa que receba como entrada uma quantia em dinheiro em reais e traduza esta quantia em palavras. O programa deve utilizar no máximo duas casas decimais e terminar quando o usuário digitar 0 como quantia. O maior valor limite do programa deve ser de R$9.999,99. Considere o seguinte exemplo de interação com o programa:

```
Digite o valor em reais (0 encerra o programa): 32.25
Valor introduzido:   32.25
O valor introduzido corresponde a: Trinta e dois reais e vinte e cinco centavos.

Digite o valor em reais (0 encerra o programa): .934
Valor introduzido:    0.93
O valor introduzido corresponde a: Noventa e tres centavos.

Digite um valor em reais: 1000000,00
Valor introduzido: 100000.00
Erro: Valor alto demais. Meu limite e' R$9999.99.

Digite o valor em reais (0 encerra o programa): 50
Valor introduzido:   50.00
O valor introduzido corresponde a: Cinquenta reais.

Digite o valor em reais (0 encerra o programa): 0
[Despedida graciosa do programa]
```

Sugestões

As sugestões para este exercício encontram-se em um documento em separado encontrado no site do livro. Não se esqueça de testar seu programa com o arquivo de dados `reais.in`, que também encontra-se no site.

EP8.23)

(a) Consulte o **Volume II** para entender o funcionamento da função **atoi()** da biblioteca padrão de C.

(b) Escreva uma função que traduza ***strings*** em números inteiros e que funcione exatamente conforme a especificação da função **atoi()**. [Obviamente, você não deverá utilizar, na implementação desta função, a própria função **atoi()**.]

(c) Escreva um programa em C que leia ***strings*** no meio de entrada padrão utilizando a função `LeString()` apresentada na **Seção 8.8** e os converta em números inteiros. O programa deve encerrar quando o usuário digitar `[ENTER]` como ***string***. Exemplo de interação com o programa:

```
Introduza uma cadeia de caracteres representando um numero
inteiro: 123
Valor convertido: 123

Introduza uma cadeia de caracteres representando um numero
inteiro: abc132
Valor convertido: 0

Introduza uma cadeia de caracteres representando um numero
inteiro: -154.65
Valor convertido: -154
Introduza uma cadeia de caracteres representando um numero
inteiro: [ENTER]
```

Sugestões

(1) A função deverá ler os caracteres (dígitos) no ***string*** e utilizá-los para construir o número correspondente. O primeiro passo desta tarefa é transformar cada dígito no valor numérico correspondente. Você não precisa ter conhecimento de tabela ASCII ou qualquer outro código específico de caracteres para fazer esta transformação. Isto é, você precisa saber apenas que, em qualquer código de caracteres, os dígitos são ordenados de `'0'` a `'9'`. Portanto, pode-se calcular o valor numérico correspondente de um dígito por meio da diferença entre o valor do dígito e o valor do dígito `'0'`. Por exemplo, `'0' - '0'` resulta em 0, `'1' - '0'` resulta em 1 e assim por diante.

(2) Sabendo o valor de cada dígito, você deverá determinar se ele corresponde a unidade, dezena, centena etc. para então multiplicá-lo pela potência de 10 correspondente. Mas como obter esta informação se, no instante em que se encontra um dígito, não se sabe ainda quantos dígitos ele tem à frente? Por exemplo, se o ***string*** for `"235"`, no instante em que se processa o dígito `'2'`, não se sabe ainda da existência dos dígitos `'3'` e `'5'`. Uma solução óbvia, mas que é trabalhosa e não é muito esperta, consiste em ir até o final do ***string*** para determinar quantos dígitos o compõem. A estratégia mais sábia é utilizar uma variável local à função para armazenar o resultado parcial da avaliação do ***string***. Esta variável seria iniciada com zero e, a cada novo dígito encontrado, multiplica-se esta variável por dez e soma-se ao valor numérico correspondente do novo dígito.

EP8.24) A **criptografia** consiste em um conjunto de técnicas utilizadas para cifrar arquivos de modo a evitar que pessoas não autorizadas tenham acesso aos conteúdos destes arquivos. A necessidade de segurança cada vez maior devido ao crescente fluxo de informações em redes de computadores tem estimulado o surgimento de algoritmos de criptografia cada vez mais sofisticados. O método mais simples de criptografia consiste na utilização de um ***código de substituição direta***. Esta 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*** de criptografia. Por exemplo, a letra `'A'` seria substituída pelo primeiro caractere (letra) no ***string*** que representa a chave, a letra `'B'` seria substituída pelo segundo caractere neste ***string*** e assim por diante. Para esta 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 iria decifrar um documento criptografado?).

a) Escreva uma função denominada `Criptografa()`, cujo protótipo é dado por:

```
char *Criptografa(char *str, char const *chave)
```

que criptografe o ***string*** passado como primeiro argumento utili-zando a chave passada como segundo argumento e retorne o ***string*** criptografado. Assuma que apenas as letras de `'a'` a `'z'` (minúsculas) são mapeadas.

b) Escreva uma função denominada `Decifra()`, cujo protótipo é dado por:

```
char *Decifra(char *str, char const *chave),
```

que decifre o ***string*** passado como primeiro argumento utilizando a chave passada como segundo argumento e retorne o ***string*** decifrado. Assuma que apenas as letras de `'a'` a `'z'` (minúsculas) foram mapeadas no ***string*** criptografado.

c) Escreva um programa que receba um ***string*** como entrada e ofereça opções para criptografá-lo ou decifrá-lo utilizando a chave:

```
tfhxqjemupidckvbaolrzwgnsy
```

(**Nota**: 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.)

O programa deve encerrar quando o usuário digitar `[RETURN]` ou `[ENTER]` como ***string***. Exemplo de interação com o programa:

```
Introduza uma linha texto: O rato roeu a roupa do rei de Roma.
Escolha C ou D para criptografar ou decifrar o texto: C
Texto criptografado:  O otrv ovqz t ovzbt xv oqu xq Rvct.

Introduza a linha texto a ser criptografada: O otrv ovqz t ovzbt xv oqu xq Rvct.
Escolha C ou D para criptografar ou decifrar o texto: D
Texto decifrado:  O rato roeu a roupa do rei de Roma.

Introduza a linha texto a ser criptografada: [ENTER]
```