6. Cadeia de caracteres

W. Celes e J. L. Rangel

6.1. Caracteres

Efetivamente, a linguagem C não oferece um tipo caractere. Os caracteres são representados por códigos numéricos. A linguagem oferece o tipo `char`, que pode armazenar valores inteiros "pequenos": um `char` tem tamanho de 1 byte, 8 bits, e sua versão com sinal pode representar valores que variam de –128 a 127. Como os códigos associados aos caracteres estão dentro desse intervalo, usamos o tipo `char` para representar caracteres[1]. A correspondência entre os caracteres e seus códigos numéricos é feita por uma tabela de códigos. Em geral, usa-se a tabela ASCII, mas diferentes máquinas podem usar diferentes códigos. Contudo, se desejamos escrever códigos portáteis, isto é, que possam ser compilados e executados em máquinas diferentes, devemos evitar o uso explícito dos códigos referentes a uma determinada tabela, como será discutido nos exemplos subseqüentes. Como ilustração, mostramos a seguir os códigos associados a alguns caracteres segundo a tabela ASCII.

Alguns caracteres que podem ser impressos (`sp` representa o branco, ou espaço):

	0	1	2	3	4	5	6	7	8	9
30			sp	!	"	#	$	%	&	'
40	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	\	]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	S	t	u	v	w
120	x	y	z	{	|	}	~			

Alguns caracteres de controle:

0	`nul`	*null*: nulo
7	`bel`	*bell*: campainha
8	`bs`	*backspace*: voltar e apagar um caractere
9	`ht`	*tab* ou tabulação horizontal
10	`nl`	*newline* ou *line feed*: mudança de linha
13	`cr`	*carriage return*: volta ao início da linha
127	`del`	*delete*: apagar um caractere

[1] Alguns alfabetos precisam de maior representatividade. O alfabeto chinês, por exemplo, tem mais de 256 caracteres, não sendo suficiente o tipo `char` (alguns compiladores oferecem o tipo `wchar`, para estes casos).

Em C, a diferença entre caracteres e inteiros é feita apenas através da maneira pela qual são tratados. Por exemplo, podemos imprimir o mesmo valor de duas formas diferentes usando formatos diferentes. Vamos analisar o fragmento de código abaixo:

```
char c = 97;
printf("%d %c\n",c,c);
```

Considerando a codificação de caracteres via tabela ASCII, a variável `c`, que foi inicializada com o valor `97`, representa o caractere `a`. A função `printf` imprime o conteúdo da variável `c` usando dois formatos distintos: com o especificador de formato para inteiro, `%d`, será impresso o valor do código numérico, `97`; com o formato de caractere, `%c`, será impresso o caractere associado ao código, a letra `a`.

Conforme mencionamos, devemos evitar o uso explícito de códigos de caracteres. Para tanto, a linguagem C permite a escrita de *constantes caracteres*. Uma constante caractere é escrita envolvendo o caractere com aspas simples. Assim, a expressão `'a'` representa uma constante caractere e resulta no valor numérico associado ao caractere `a`. Podemos, então, reescrever o fragmento de código acima sem particularizar a tabela ASCII.

```
char c = 'a';
printf("%d %c\n", c, c);
```

Além de agregar portabilidade e clareza ao código, o uso de constantes caracteres nos livra de conhecermos os códigos associados a cada caractere.

Independente da tabela de códigos numéricos utilizada, garante-se que os dígitos são codificados em seqüência. Deste modo, se o dígito *zero* tem código 48, o dígito *um* tem obrigatoriamente código 49, e assim por diante. As letras minúsculas e as letras maiúsculas também formam dois grupos de códigos seqüenciais. O exemplo a seguir tira proveito desta seqüência dos códigos de caracteres.

Exemplo. Suponhamos que queremos escrever uma função para testar se um caractere `c` é um *dígito* (um dos caracteres entre `'0'` e `'9'`). Esta função pode ter o protótipo:

```
int digito(char c);
```

e ter como resultado `1` (verdadeiro) se `c` for um dígito, e `0` (falso) se não for.

A implementação desta função pode ser dada por:

```
int digito(char c)
{
   if ((c>='0')&&(c<='9'))
      return 1;
   else
      return 0;
}
```

Exercício. Escreva uma função para determinar se um caractere é uma letra, com protótipo:

```
int letra(char c);
```

Exercício. Escreva uma função para converter um caractere para maiúscula. Se o caractere dado representar uma letra minúscula, devemos ter como valor de retorno a letra maiúscula correspondente. Se o caractere dado não for uma letra minúscula, devemos ter como valor de retorno o mesmo caractere, sem alteração. O protótipo desta função pode ser dado por:

```
char maiuscula(char c);
```

6.2. Cadeia de caracteres (strings)

Cadeias de caracteres (*strings*), em C, são representadas por vetores do tipo `char` terminadas, *obrigatoriamente*, pelo caractere nulo ('\0'). Portanto, para armazenarmos uma cadeia de caracteres, devemos reservar uma posição adicional para o caractere de fim da cadeia. Todas as funções que manipulam cadeias de caracteres (e a biblioteca padrão de C oferece várias delas) recebem como parâmetro um vetor de `char`, isto é, um ponteiro para o primeiro elemento do vetor que representa a cadeia, e processam caractere por caractere, até encontrarem o caractere nulo, que sinaliza o final da cadeia.

Por exemplo, o especificador de formato `%s` da função `printf` permite imprimir uma cadeia de caracteres. A função `printf` então recebe um vetor de `char` e imprime elemento por elemento, até encontrar o caractere nulo.

O código abaixo ilustra a representação de uma cadeia de caracteres. Como queremos representar a palavra `Rio`, composta por 3 caracteres, declaramos um vetor com dimensão 4 (um elemento adicional para armazenarmos o caractere nulo no final da cadeia). O código preenche os elementos do vetor, incluindo o caractere `'\0'`, e imprime a palavra na tela.

```
int main ( void )
{
   char cidade[4];
   cidade[0] = 'R';
   cidade[1] = 'i';
   cidade[2] = 'o';
   cidade[3] = '\0';
   printf("%s \n", cidade);
   return 0;
}
```

Se o caractere `'\0'` não fosse colocado, a função `printf` executaria de forma errada, pois não conseguiria identificar o final da cadeia.

Como as cadeias de caracteres são vetores, podemos reescrever o código acima inicializando os valores dos elementos do vetor na declaração:

```
int main ( void )
```

```
{
   char cidade[ ] = {'R', 'i', 'o', '\0'};
   printf("%s \n", cidade);
   return 0;
}
```

A inicialização de cadeias de caracteres é tão comum em códigos C que a linguagem permite que elas sejam inicializadas escrevendo-se os caracteres entre aspas duplas. Neste caso, o caractere nulo é representado implicitamente. O código acima pode ser reescrito da seguinte forma:

```
int main ( void )
{
   char cidade[ ] = "Rio";
   printf("%s \n", cidade);
   return 0;
}
```

A variável `cidade` é automaticamente dimensionada e inicializada com 4 elementos. Para ilustrar a declaração e inicialização de cadeias de caracteres, consideremos as declarações abaixo:

```
char s1[] = "";
char s2[] = "Rio de Janeiro";
char s3[81];
char s4[81] = "Rio";
```

Nestas declarações, a variável `s1` armazena uma cadeia de caracteres vazia, representada por um vetor com um único elemento, o caractere `'\0'`. A variável `s2` representa um vetor com `15` elementos. A variável `s3` representa uma cadeia de caracteres capaz de representar cadeias com até `80` caracteres, já que foi dimensionada com `81` elementos. Esta variável, no entanto, não foi inicializada e seu conteúdo é desconhecido. A variável `s4` também foi dimensionada para armazenar cadeias até `80` caracteres, mas seus primeiros quatro elementos foram atribuídos na declaração.

Leitura de caracteres e cadeias de caracteres

Para capturarmos o valor de um caractere simples fornecido pelo usuário via teclado, usamos a função `scanf`, com o especificador de formato `%c`.

```
char a;
...
scanf("%c", &a);
...
```

Desta forma, se o usuário digitar a letra `r`, por exemplo, o código associado à letra `r` será armazenado na variável `a`. Vale ressaltar que, diferente dos especificadores `%d` e `%f`, o especificador `%c` não pula os caracteres brancos[2]. Portanto, se o usuário teclar um espaço

[2] Um "caractere branco" pode ser um espaço (`' '`), um caractere de tabulação (`'\t'`) ou um caractere de nova linha (`'\n'`).

antes da letra `r`, o código do espaço será capturado e a letra `r` será capturada apenas numa próxima chamada da função `scanf`. Se desejarmos pular todas as ocorrências de caracteres brancos que porventura antecedam o caractere que queremos capturar, basta incluir um espaço em branco no formato, antes do especificador.

```
char a;
...
scanf(" %c", %a);   /* o branco no formato pula brancos da entrada */
...
```

Já mencionamos que o especificador `%s` pode ser usado na função `printf` para imprimir uma cadeia de caracteres. O mesmo especificador pode ser utilizado para capturar cadeias de caracteres na função `scanf`. No entanto, seu uso é muito limitado. O especificador `%s` na função `scanf` pula os eventuais caracteres brancos e captura a seqüência de caracteres não brancos. Consideremos o fragmento de código abaixo:

```
char cidade[81];
...
scanf("%s", cidade);
...
```

Devemos notar que não usamos o caractere `&` na passagem da cadeia para a função, pois a cadeia é um vetor (o nome da variável representa o endereço do primeiro elemento do vetor e a função atribui os valores dos elementos a partir desse endereço). O uso do especificador de formato `%s` na leitura é limitado, pois o fragmento de código acima funciona apenas para capturar nomes simples. Se o usuário digitar `Rio de Janeiro`, apenas a palavra `Rio` será capturada, pois o `%s` lê somente uma seqüência de caracteres não brancos.

Em geral, queremos ler nomes compostos (nome de pessoas, cidades, endereços para correspondência, etc.). Para capturarmos estes nomes, podemos usar o especificador de formato `%[...]`, no qual listamos entre os colchetes todos os caracteres que aceitaremos na leitura. Assim, o formato `"%[aeiou]"` lê seqüências de vogais, isto é, a leitura prossegue até que se encontre um caractere que não seja uma vogal. Se o primeiro caractere entre colchetes for o acento circunflexo (`^`), teremos o efeito inverso (negação). Assim, com o formato `"%[^aeiou]"` a leitura prossegue enquanto uma vogal não for encontrada. Esta construção permite capturarmos nomes compostos. Consideremos o código abaixo:

```
char cidade[81];
...
scanf(" %[^\n]", cidade);
...
```

A função `scanf` agora lê uma seqüência de caracteres até que seja encontrado o caractere de mudança de linha (`'\n'`). Em termos práticos, captura-se a linha fornecida pelo usuário até que ele tecle "*Enter*". A inclusão do espaço no formato (antes do sinal `%`) garante que eventuais caracteres brancos que precedam o nome serão pulados.

Para finalizar, devemos salientar que o trecho de código acima é perigoso, pois, se o usuário fornecer uma linha que tenha mais de 80 caracteres, estaremos invadindo um espaço de memória que não está reservado (o vetor foi dimensionado com 81 elementos).

Para evitar esta possível invasão, podemos limitar o número máximo de caracteres que serão capturados.

```
char cidade[81];
...
scanf(" %80[^\n]", cidade);      /* lê no máximo 80 caracteres */
...
```

Exemplos de funções que manipulam cadeias de caracteres

Nesta seção, discutiremos a implementação de algumas funções que manipulam cadeias de caracteres.

Exemplo. Impressão caractere por caractere.
Vamos inicialmente considerar a implementação de uma função que imprime uma cadeia de caracteres, caractere por caractere. A implementação pode ser dada por:

```
void imprime (char* s)
{
   int i;
   for (i=0; s[i] != '\0'; i++)
      printf("%c",s[i]);
   printf("\n");
}
```

que teria funcionalidade análoga à utilização do especificador de formato `%s`.

```
void imprime (char* s)
{
   printf("%s\n",s);
}
```

Exemplo. Comprimento da cadeia de caracteres.
Consideremos a implementação de uma função que recebe como parâmetro de entrada uma cadeia de caracteres e fornece como retorno o número de caracteres existentes na cadeia. O protótipo da função pode ser dado por:

```
int comprimento (char* s);
```

Para contar o número de caracteres da cadeia, basta contarmos o número de caracteres até que o caractere nulo (que indica o fim da cadeia) seja encontrado. O caractere nulo em si não deve ser contado. Uma possível implementação desta função é:

```
int comprimento (char* s)
{
   int i;
   int n = 0;  /* contador */
   for (i=0; s[i] != '\0'; i++)
      n++;
   return n;
}
```

O trecho de código abaixo faz uso da função acima.

```
#include <stdio.h>

int comprimento (char* s);

int main (void)
{
   int tam;
   char cidade[] = "Rio de Janeiro";
   tam = comprimento(cidade);
   printf("A string \"%s\" tem %d caracteres\n", cidade, tam);
   return 0;
}
```

A saída deste programa será: `A string "Rio de Janeiro" tem 14 caracteres`. Salientamos o uso do caractere de escape `\"` para incluir as aspas na saída.

Exemplo. Cópia de cadeia de caracteres.
Vamos agora considerar a implementação de uma função para copiar os elementos de uma cadeia de caracteres para outra. Assumimos que a cadeia que receberá a cópia tem espaço suficiente para que a operação seja realizada. O protótipo desta função pode ser dado por:

```
void copia (char* dest, char* orig);
```

A função copia os elementos da cadeia original (`orig`) para a cadeia de destino (`dest`). Uma possível implementação desta função é mostrada abaixo:

```
void copia (char* dest, char* orig)
{
   int i;
   for (i=0; orig[i] != '\0'; i++)
      dest[i] = orig[i];
   /* fecha a cadeia copiada */
   dest[i] = '\0';
}
```

Salientamos a necessidade de "fechar" a cadeia copiada após a cópia dos caracteres não nulos. Quando o laço do `for` terminar, a variável `i` terá o índice de onde está armazenado o caractere nulo na cadeia original. A cópia também deve conter o `'\0'` nesta posição.

Exemplo. Concatenação de cadeias de caracteres.
Vamos considerar uma extensão do exemplo anterior e discutir a implementação de uma função para concatenar uma cadeia de caracteres com outra já existente. Isto é, os caracteres de uma cadeia são copiados no final da outra cadeia. Assim, se uma cadeia representa inicialmente a cadeia `PUC` e concatenarmos a ela a cadeia `Rio`, teremos como resultado a cadeia `PUCRio`. Vamos mais uma vez considerar que existe espaço reservado que permite fazer a cópia dos caracteres. O protótipo da função pode ser dado por:

```
void concatena (char*dest, char* orig);
```

Uma possível implementação desta função é mostrada a seguir:

```
void concatena (char*dest, char* orig)
{
   int i = 0; /* indice usado na cadeia destino, inicializado com zero */
   int j;     /* indice usado na cadeia origem */
   /* acha o final da cadeia destino */
   i = 0;
   while (s[i] != '\0')
      i++;
   /* copia elementos da origem para o final do destino */
   for (j=0; orig[j] != '\0'; j++)
   {
      dest[i] = orig[j];
      i++;
   }
   /* fecha cadeia destino */
   dest[i] = '\0';
}
```

Funções análogas às funções `comprimento`, `copia` e `concatena` são disponibilizadas pela biblioteca padrão de C. As funções da biblioteca padrão são, respectivamente, `strlen`, `strcpy` e `strcat`, que fazem parte da biblioteca de cadeias de caracteres (*strings*), `string.h`. Existem diversas outras funções que manipulam cadeias de caracteres nessa biblioteca. A razão de mostrarmos possíveis implementações destas funções como exercício é ilustrar a codificação da manipulação de cadeias de caracteres.

Exemplo 5: Duplicação de cadeias de caracteres.
Consideremos agora um exemplo com alocação dinâmica. O objetivo é implementar uma função que receba como parâmetro uma cadeia de caracteres e forneça uma cópia da cadeia, alocada dinamicamente. O protótipo desta função pode ser dado por:

```
char* duplica (char* s);
```

Uma possível implementação, usando as funções da biblioteca padrão, é:

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

char* duplica (char* s)
{
   int n = strlen(s);
   char* d = (char*) malloc ((n+1)*sizeof(char));
   strcpy(d,s);
   return d;
}
```

A função que chama `duplica` fica responsável por liberar o espaço alocado.

Funções recursivas

Uma cadeia de caracteres pode ser definida de forma recursiva. Podemos dizer que uma cadeia de caracteres é representada por:

- uma cadeia de caracteres vazia; ou
- um caractere seguido de uma (sub) cadeia de caracteres.

Isto é, podemos dizer que uma cadeia `s` não vazia pode ser representada pelo seu primeiro caractere `s[0]` seguido da cadeia que começa no endereço do então segundo caractere, `&s[1]`.

Vamos reescrever algumas das funções mostradas acima, agora com a versão recursiva.

Exemplo. Impressão caractere por caractere.
Uma versão recursiva da função para imprimir a cadeia caractere por caractere é mostrada a seguir. Como já foi discutido, uma implementação recursiva deve ser projetada considerando-se a definição recursiva do objeto, no caso uma cadeia de caracteres. Assim, a função deve primeiro testar se a condição da cadeia é vazia. Se a cadeia for vazia, nada precisa ser impresso; se não for vazia, devemos imprimir o primeiro caractere e então chamar uma função para imprimir a sub-cadeia que se segue. Para imprimir a sub-cadeia podemos usar a própria função, recursivamente.

```
void imprime_rec (char* s)
{
   if (s[0] != '\0')
   {
      printf("%c",s[0]);
      imprime_rec(&s[1]);
   }
}
```

Algumas implementações ficam bem mais simples se feitas recursivamente. Por exemplo, é simples alterar a função acima e fazer com que os caracteres da cadeia sejam impressos em ordem inversa, de trás para a frente: basta imprimir a sub-cadeia antes de imprimir o primeiro caractere.

```
void imprime_inv (char* s)
{
   if (s[0] != '\0')
   {
      imprime_inv(&s[1]);
      printf("%c",s[0]);
   }
}
```

Como exercício, sugerimos implementar a impressão inversa sem usar recursividade.

Exemplo. Comprimento da cadeia de caracteres.
Uma implementação recursiva da função que retorna o número de caracteres existentes na cadeia é mostrada a seguir:

```
int comprimento_rec (char* s)
{
   if (s[0] == '\0')
      return 0;
   else
      return 1 + comprimento_rec(&s[1]);
}
```

Exemplo. Cópia de cadeia de caracteres.
Vamos mostrar agora uma possível implementação recursiva da função `copia` mostrada anteriormente.

```
void copia_rec (char* dest, char* orig)
{
   if (orig[0] == '\0')
      dest[0] = '\0';
   else {
      dest[0] = orig[0];
      copia_rec(&dest[1],&orig[1]);
   }
}
```

É fácil verificar que o código acima pode ser escrito de forma mais compacta:

```
void copia_rec_2 (char* dest, char* orig)
{
   dest[0] = orig[0];
   if (orig[0] != '\0')
      copia_rec_2(&dest[1],&orig[1]);
}
```

Constante cadeia de caracteres**

Em códigos C, uma seqüência de caracteres delimitada por aspas representa uma constante cadeia de caracteres, ou seja, uma expressão constante, cuja avaliação resulta no ponteiro onde a cadeia de caracteres está armazenada. Para exemplificar, vamos considerar o trecho de código abaixo:

```
#include <string.h>

int main ( void )
{
   char cidade[4];
   strcpy (cidade, "Rio" );
   printf ( "%s \n", cidade );
   return 0;
}
```

De forma ilustrativa, o que acontece é que, quando o compilador encontra a cadeia `"Rio"`, automaticamente é alocada na área de constantes a seguinte seqüência de caracteres:

```
'R', 'i', 'o', '\0'
```

e é fornecido o ponteiro para o primeiro elemento desta seqüência. Assim, a função `strcpy` recebe dois ponteiros de cadeias: o primeiro aponta para o espaço associado à variável `cidade` e o segundo aponta para a área de constantes onde está armazenada a cadeia `Rio`.

Desta forma, também é válido escrever:

```
int main (void)
{
   char *cidade;    /* declara um ponteiro para char */
   cidade = "Rio";  /* cidade recebe o endereco da cadeia "Rio" */
   printf ( "%s \n", cidade );
   return 0;
}
```

Existe uma diferença sutil entre as duas declarações abaixo:

```
char s1[] = "Rio de Janeiro";
char* s2 = "Rio de Janeiro";
```

Na primeira, declaramos um vetor de `char` local que é inicializado com a cadeia de caracteres `Rio de Janeiro`, seguido do caractere nulo. A variável `s1` ocupa, portanto, 15 bytes de memória. Na segunda, declaramos um ponteiro para `char` que é inicializado com o endereço de uma área de memória onde a constante cadeia de caracteres `Rio de Janeiro` está armazenada. A variável `s2` ocupa 4 bytes (espaço de um ponteiro). Podemos verificar esta diferença imprimindo os valores `sizeof(s1)` e `sizeof(s2)`. Como `s1` é um vetor local, podemos alterar o valor de seus elementos. Por exemplo, é válido escrever `s1[0]='X';` alterando o conteúdo da cadeia para `Xio de Janeiro`. No entanto, não é válido escrever `s2[0]='X';` pois estaríamos tentando alterar o conteúdo de uma área de constante.

6.3. Vetor de cadeia de caracteres

Em muitas aplicações, desejamos representar um vetor de cadeia de caracteres. Por exemplo, podemos considerar uma aplicação que armazene os nomes de todos os alunos de uma turma num vetor. Sabemos que uma cadeia de caracteres é representada por um vetor do tipo `char`. Para representarmos um vetor onde cada elemento é uma cadeia de caracteres, devemos ter um vetor cujos elementos são do tipo `char*`, isto é, um vetor de ponteiros para `char`. Assim, criamos um conjunto (vetor) bidimensional de `char`. Assumindo que o nome de nenhum aluno terá mais do que 80 caracteres e que o número máximo de alunos numa turma é 50, podemos declarar um vetor bidimensional para armazenar os nomes dos alunos

```
char alunos[50][81];
```

Com esta variável declarada, `alunos[i]` acessa a cadeia de caracteres com o nome do `(i+1)`-ésimo aluno da turma e, conseqüentemente, `alunos[i][j]` acessa a `(j+1)`-ésima letra do nome do `(i+1)`-ésimo aluno. Considerando que `alunos` é uma variável global, uma função para imprimir os nomes dos `n` alunos de uma turma poderia ser dada por:

```
void imprime (int n)
{
   int i;
   for (i=0; i<n; i++)
      printf("%s\n", alunos[i]);
}
```

No próximo capítulo, que trata de matrizes, discutiremos conjuntos bidimensionais com mais detalhes. Para a representação de vetores de cadeias de caracteres, optamos, em geral, por declarar um vetor de ponteiros e alocar dinamicamente cada elemento (no caso, uma cadeia de caracteres). Desta forma, otimizamos o uso do espaço de memória, pois não precisamos achar uma dimensão máxima para todas as cadeias do vetor nem desperdiçamos espaço excessivo quando temos poucos nomes de alunos a serem armazenados. Cada elemento do vetor é um ponteiro. Se precisarmos armazenar um nome na posição, alocamos o espaço de memória necessário para armazenar a cadeia de caracteres correspondente. Assim, nosso vetor com os nomes dos alunos pode ser declarado da seguinte forma:

```
#define MAX 50
char* alunos[MAX];
```

Exemplo. Leitura e impressão dos nomes dos alunos.
Vamos escrever uma função que captura os nomes dos alunos de uma turma. A função inicialmente lê o número de alunos da turma (que deve ser menor ou igual a `MAX`) e captura os nomes fornecidos por linha, fazendo a alocação correspondente. Para escrever esta função, podemos pensar numa função auxiliar que captura uma linha e fornece como retorno uma cadeia alocada dinamicamente com a linha inserida. Fazendo uso das funções que escrevemos acima, podemos ter:

```
char* lelinha (void)
{
   char linha[121];       /* variavel auxiliar para ler linha */
   scanf(" %120[^\n]",linha);
   return duplica(linha);
}
```

A função para capturar os nomes dos alunos preenche o vetor de nomes e pode ter como valor de retorno o número de nomes lidos:

```
int lenomes (char** alunos)
{
   int i;
   int n;
   do {
      scanf("%d",&n);
   } while (n>MAX);

   for (i=0; i<n; i++)
      alunos[i] = lelinha();
   return n;
}
```

A função para liberar os nomes alocados na tabela pode ser implementada por:

```
void liberanomes (int n, char** alunos)
{
   int i;
   for (i=0; i<n; i++)
      free(alunos[i]);
}
```

Uma função para imprimir os nomes dos alunos pode ser dada por:

```
void imprimenomes (int n, char** alunos)
{
   int i;
   for (i=0; i<n; i++)
      printf("%s\n", alunos[i]);
}
```

Um programa que faz uso destas funções é mostrado a seguir:

```
int main (void)
{
   char* alunos[MAX];
   int n = lenomes(alunos);
   imprimenomes(n,alunos);
   liberanomes(n,alunos);
   return 0;
}
```

Parâmetros da função `main`**

Em todos os exemplos mostrados, temos considerado que a função principal, `main`, não recebe parâmetros. Na verdade, a função `main` pode ser definida para receber zero ou dois parâmetros, geralmente chamados `argc` e `argv`. O parâmetro `argc` recebe o número de argumentos passados para o programa quando este é executado; por exemplo, de um comando de linha do sistema operacional. O parâmetro `argv` é um vetor de cadeias de caracteres, que armazena os nomes passados como argumentos. Por exemplo, se temos um programa executável com o nome `mensagem` e se ele for invocado através da linha de comando:

```
> mensagem estruturas de dados
```

a variável `argc` receberá o valor `4` e o vetor `argv` será inicializado com os seguintes elementos: `argv[0]="mensagem"`, `argv[1]="estruturas"`, `argv[2]="de"`, e `argv[3]="dados"`. Isto é, o primeiro elemento armazena o próprio nome do executável e os demais são preenchidos com os nomes passados na linha de comando. Esses parâmetros podem ser úteis para, por exemplo, passar o nome de um arquivo de onde serão capturados os dados de um programa. A manipulação de arquivos será discutida mais adiante no curso. Por ora, mostramos um exemplo simples que trata os dois parâmetros da função `main`.

```
#include <stdio.h>
int main (int argc, char** argv)
{
   int i;
   for (i=0; i<argc; i++)
      printf("%s\n", argv[i]);
   return 0;
}
```

Se este programa tiver seu executável chamado de `mensagem` e for invocado com a linha de comando mostrada acima, a saída será:

```
mensagem
estruturas
de
dados
```