AULA 14

PONTEIROS E CADEIAS

Nas aulas 12 e 13 estudamos formas de trabalhar com variáveis compostas homogêneas e ponteiros para seus elementos. No entanto, é importante ainda estudar a relação entre ponteiros e as cadeias de caracteres que, como já vimos, são vetores especiais que contêm caracteres. Nesta aula, baseada na referência [7], aprenderemos algumas particularidades de ponteiros para elementos de cadeias de caracteres na linguagem C, além de estudar a relação entre ponteiros e constantes que são cadeias de caracteres.

14.1 Literais e ponteiros

Devemos relembrar que uma **literal** é uma seqüência de caracteres envolvida por aspas duplas. Um exemplo de uma literal é apresentado a seguir[1]:

```
"O usuário médio de computador possui o cérebro de um macaco-aranha"
```

Nosso primeiro contato com literais foi ainda em Algoritmos e Programação I. Literais ocorrem com freqüência na chamada das funções `printf` e `scanf`. Mas quando chamamos uma dessas funções e fornecemos uma literal como argumento, o que de fato estamos passando? Em essência, a linguagem C trata literais como cadeias de caracteres. Quando o compilador encontra uma literal de comprimento n em um programa, ele reserva um espaço de $n+1$ bytes na memória. Essa área de memória conterá os caracteres da literal mais o caracter nulo que indica o final da cadeia. O caracter nulo é um byte cujos bits são todos zeros e é representado pela seqüência de caracteres `\0`. Por exemplo, a literal `"abc"` é armazenada como uma cadeia de quatro caracteres, como mostra a figura 14.1.

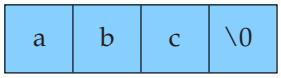

Figura 14.1: Representação de uma literal na memória.

Literais podem ser vazias. A literal `""` é armazenada como um único caractere nulo.

Como uma literal é armazenada em um vetor, o compilador a enxerga como um ponteiro do tipo `char *`. As funções `printf` e `scanf`, por exemplo, esperam um valor do tipo `char *` como primeiro argumento. Se, por exemplo, fazemos a seguinte chamada:

[1] Frase de Bill Gates, dono da Micro$oft, em uma entrevista para *Computer Magazine*.

```
printf("abc");
```

o endereço da literal `"abc"` é passado como argumento para a função `printf`, isto é, o endereço de onde se encontra o caractere `a` na memória.

Em geral, podemos usar uma literal sempre que a linguagem C permita o uso de um ponteiro do tipo `char *`. Por exemplo, uma literal pode ocorrer do lado direito de uma atribuição, como mostrado a seguir:

```
char *p;
p = "abc";
```

Essa atribuição não copia os caracteres de `"abc"`, mas faz o ponteiro p apontar para o primeiro caractere da literal, como mostra a figura 14.2.

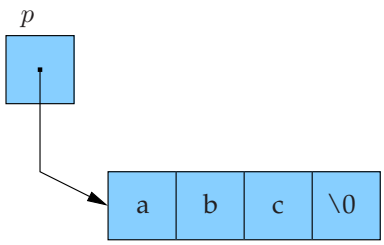

Figura 14.2: Representação da atribuição de uma literal a um ponteiro.

Observe ainda que não é permitido alterar uma literal durante a execução de um programa. Isso significa que a tentativa de modificar uma literal pode causar um comportamento indefinido do programa.

Relembrando, uma variável cadeia de caracteres é um vetor do tipo `char` que necessariamente deve reservar espaço para o caractere nulo. Quando declaramos um vetor de caracteres que será usado para armazenar cadeias de caracteres, devemos sempre declarar esse vetor com uma posição a mais que a mais longa das cadeias de caracteres possíveis, já que por convenção da linguagem C, toda cadeia de caracteres é finalizada com um caractere nulo. Veja, por exemplo, a declaração a seguir:

```
char cadeia[TAM+1];
```

onde `TAM` é uma macro definida com o tamanho da cadeia de caracteres mais longa que pode ser armazenada na variável `cadeia`.

Lembrando ainda, podemos inicializar uma cadeia de caracteres no momento de sua declaração, como mostra o exemplo abaixo:

```
char data[13] = "7 de outubro";
```

O compilador então coloca os caracteres de `"7 de outubro"` no vetor `data` e adiciona o caractere nulo ao final para que `data` possa ser usada como uma cadeia de caracteres. A figura 14.3 ilustra essa situação.

Figura 14.3: Declaração e inicialização de uma cadeia de caracteres.

Apesar de `"7 de outubro"` se parecer com uma literal, a linguagem C de fato a vê como uma abreviação para um inicializador de um vetor. Ou seja, a declaração e inicialização acima é enxergada pelo compilador como abaixo:

```
char data[13] = {'7',' ','d','e',' ','o','u','t','u','b','r','o','\0'};
```

No caso em que o inicializador é menor que o tamanho definido para a cadeia de caracteres, o compilador preencherá as posições finais restantes da cadeia com o caractere nulo. Por outro lado, é sempre importante garantir que o inicializador tenha menor comprimento que o tamanho do vetor declarado. Também, podemos omitir o tamanho do vetor em uma declaração e inicialização simultâneas, caso em que o vetor terá o tamanho equivalente ao comprimento do inicializador mais uma unidade, que equivale ao caractere nulo.

Agora, vamos comparar a declaração abaixo:

```
char data[] = "7 de outubro";
```

que declara um vetor `data`, que é uma cadeia de caracteres, com a declaração a seguir:

```
char *data = "7 de outubro";
```

que declara `data` como um ponteiro. Devido à relação estrita entre vetores e ponteiros que vimos na aula 12, podemos usar as duas versões da declaração de `data` como uma cadeia de caracteres. Em particular, qualquer função que receba um vetor/cadeia de caracteres ou um ponteiro para caracteres aceita qualquer uma das versões da declaração da variável `data` apresentada acima.

No entanto, devemos ter cuidado para não cometer o erro de acreditar que as duas versões da declaração de `data` são equivalentes e intercambiáveis. Existem diferenças significativas entre as duas, que destacamos abaixo:

- na versão em que a variável é declarada como um vetor, os caracteres armazenados em `data` podem ser modificados, como fazemos com elementos de qualquer vetor; na versão em que a variável é declarada como um ponteiro, `data` aponta para uma literal que, como já vimos, não pode ser modificada;

- na versão com vetor, `data` é um identificador de um vetor; na versão com ponteiro, `data` é uma variável que pode, inclusive, apontar para outras cadeias de caracteres durante a execução do programa.

Se precisamos que uma cadeia de caracteres seja modificada, é nossa responsabilidade declarar um vetor de caracteres no qual será armazenada essa cadeia. Declarar um ponteiro não é suficiente, neste caso. Por exemplo, a declaração abaixo:

```
char *p;
```

faz com que o compilador reserve espaço suficiente para uma variável ponteiro. Infelizmente, o compilador não reserva espaço para uma cadeia de caracteres, mesmo porque, não há indicação alguma de um possível comprimento da cadeia de caracteres que queremos armazenar. Antes de usarmos p como uma cadeia de caracteres, temos de fazê-la apontar para um vetor de caracteres. Uma possibilidade é fazer p apontar para uma variável que é uma cadeia de caracteres, como mostramos a seguir:

```
char cadeia[TAM+1], *p;
p = cadeia;
```

Com essa atribuição, p aponta para o primeiro caractere de `cadeia` e assim podemos usar p como uma cadeia de caracteres. Outra possibilidade é fazer p apontar para uma cadeia de caracteres dinamicamente alocada, como veremos na aula 16.

Ainda poderíamos discorrer sobre processos para leitura e escrita de cadeias de caracteres, sobre acesso aos caracteres de uma cadeia de caracteres e também sobre o uso das funções da biblioteca da linguagem C que trabalham especificamente com cadeias de caracteres, o que já fizemos nas aulas de Algoritmos e Programação I. Ainda veremos a seguir dois tópicos importantes sobre cadeias de caracteres: vetores de cadeias de caracteres e argumentos de linha de comando.

14.2 Vetores de cadeias de caracteres

Uma forma de armazenar em memória um vetor de cadeias de caracteres é através da criação de uma matriz de caracteres e então armazenar as cadeias de caracteres uma a uma. Por exemplo, podemos fazer como a seguir:

```
char planetas[][9] = {"Mercurio", "Venus", "Terra",
                      "Marte", "Jupiter", "Saturno",
                      "Urano", "Netuno", "Plutao"};
```

Observe que estamos omitindo o número de linhas da matriz, que é fornecido pelo inicializador, mas a linguagem C exige que o número de colunas seja especificado, conforme fizemos na declaração.

A figura 14.4 ilustra a declaração e inicialização da variável `planetas`. Observe que todas as cadeias cabem nas colunas da matriz e, também, que há um tanto de compartimentos desperdiçados na matriz, preenchidos com o caractere `\0`, já que nem todas as cadeias são compostas por 8 caracteres.

	0	1	2	3	4	5	6	7	8
0	M	e	r	c	u	r	i	o	\0
1	V	e	n	u	s	\0	\0	\0	\0
2	T	e	r	r	a	\0	\0	\0	\0
3	M	a	r	t	e	\0	\0	\0	\0
4	J	u	p	i	t	e	r	\0	\0
5	S	a	t	u	r	n	o	\0	\0
6	U	r	a	n	o	\0	\0	\0	\0
7	N	e	t	u	n	o	\0	\0	\0
8	P	l	u	t	a	o	\0	\0	\0

Figura 14.4: Matriz de cadeias de caracteres `planetas`.

A ineficiência de armazenamento aparente nesse exemplo é comum quando trabalhamos com cadeias de caracteres, já que coleções de cadeias de caracteres serão, em geral, um misto entre curtas e longas cadeias. Uma possível forma de sanar esse problema é usar um vetor cujos elementos são ponteiros para cadeias de caracteres, como podemos ver na declaração abaixo:

```
char *planetas[] = {"Mercurio", "Venus", "Terra",
                    "Marte", "Jupiter", "Saturno",
                    "Urano", "Netuno", "Plutao"};
```

Note que há poucas diferenças entre essa declaração e a declaração anterior da variável `planetas`: removemos um par de colchetes com um número no interior deles e colocamos um asterisco precedendo o identificador da variável. No entanto, o efeito dessa declaração na memória é muito diferente, como podemos ver na figura 14.5.

Cada elemento do vetor `planetas` é um ponteiro para uma cadeia de caracteres, terminada com um caractere nulo. Não há mais desperdício de compartimentos nas cadeias de caracteres, apesar de termos de alocar espaço para os ponteiros no vetor `planetas`. Para acessar

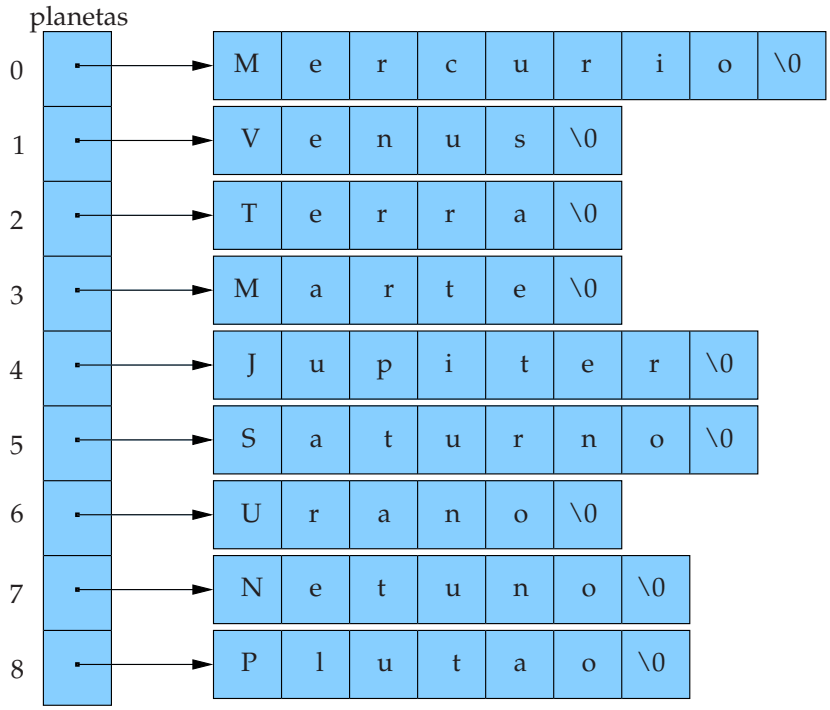

Figura 14.5: Vetor `planetas` de ponteiros para cadeias de caracteres.

um dos nomes dos planetas necessitamos apenas do índice do vetor. Para acessar um caractere do nome de um planeta devemos fazer da mesma forma como acessamos um elemento em uma matriz. Por exemplo, para buscar cadeias de caracteres no vetor `planetas` que iniciam com a letra M, podemos usar o seguinte trecho de código:

```
for (i = 0; i < 9; i++)
   if (planetas[i][0] == 'M')
      printf("%s começa com M\n", planetas[i]);
```

14.3 Argumentos na linha de comandos

Quando executamos um programa, em geral, devemos fornecer a ele alguma informação como, por exemplo, um nome de um arquivo, uma opção que modifica seu comportamento, etc. Por exemplo, considere o comando UNIX `ls`. Se executamos esse comando como abaixo:

```
prompt$ ls
```

em uma linha de comando, o resultado será uma listagem de nomes dos arquivos no diretório atual. Se digitamos o comando seguido de uma opção, como abaixo:

```
prompt$ ls -l
```

então o resultado é uma listagem detalhada[2] que nos mostra o tamanho de cada arquivo, seu proprietário, a data e hora em que houve a última modificação no arquivo e assim por diante. Para modificar ainda mais o comportamento do comando `ls` podemos especificar que ele mostre detalhes de apenas um arquivo, como mostrado abaixo:

```
prompt$ ls -l exerc1.c
```

Nesse caso, o comando `ls` mostrará informações detalhadas sobre o arquivo `exerc1.c`.

Informações em linha de comando estão disponíveis para todos os programas, não apenas para comandos do sistema operacional. Para ter acesso aos **argumentos de linha de comando**, chamados de **parâmetros do programa** na linguagem C padrão, devemos definir a função `main` como uma função com dois parâmetros que costumeiramente têm identificadores `argc` e `argv`. Isto é, devemos fazer como abaixo:

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

O parâmetro `argc`, abreviação de "contador de argumentos", é o número de argumentos de linha de comando, incluindo também o nome do programa. O parâmetro `argv`, abreviação de "vetor de argumentos", é um vetor de ponteiros para os argumentos da linha de comando, que são armazenados como cadeias de caracteres. Assim, `argv[0]` aponta para o nome do programa, enquanto que `argv[1]` até `argv[argc-1]` apontam para os argumentos da linha de comandos restantes. O vetor `argv` tem um elemento adicional `argv[argc]` que é sempre um ponteiro nulo, um ponteiro especial que aponta para nada, representado pela macro `NULL`.

Se um(a) usuário(a) digita a linha de comando abaixo:

```
prompt$ ls -l exerc1.c
```

então `argc` conterá o valor 3, `argv[0]` apontará para a cadeia de caracteres com o nome do programa, `argv[1]` apontará para a cadeia de caracteres `"-l"`, `argv[2]` apontará para a cadeia de caracteres `"exerc1.c"` e `argv[3]` apontará para nulo. A figura 14.6 ilustra essa situação. Observe que o nome do programa não foi listado porque pode incluir o nome do diretório ao qual o programa pertence ou ainda outras informações que dependem do sistema operacional.

[2] `l` do inglês *long*.

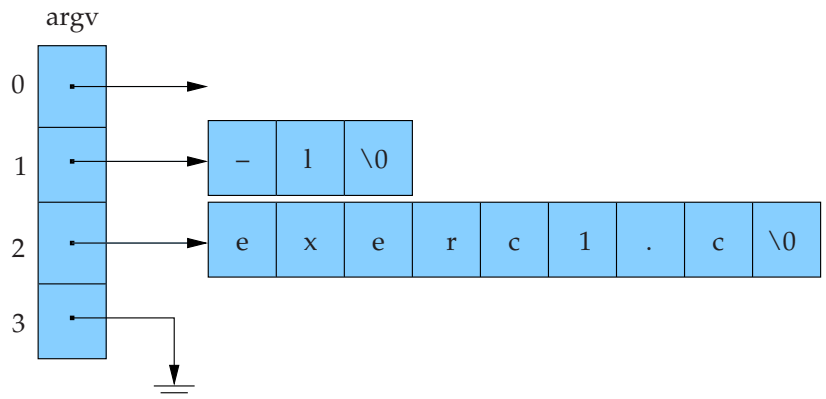

Figura 14.6: Representação de `argv`.

Como `argv` é um vetor de ponteiros, o acesso aos argumentos da linha de comandos é realizado, em geral, como mostrado na estrutura de repetição a seguir:

```
int i;
⋮
for (i = 1; i < argc; i++)
   printf("%s\n", argv[i]);
```

O programa 14.1 ilustra como acessar os argumentos de uma linha de comandos.

Programa 14.1: Verifica nomes de planetas.

```
#include <stdio.h>
#include <string.h>
#define NUM_PLANETAS 9

int main(int argc, char *argv[])
{
   char *planetas[] = {"Mercurio", "Venus", "Terra", "Marte", "Jupiter",
                       "Saturno", "Urano", "Netuno", "Plutao"};
   int i, j, k;

   for (i = 1; i < argc; i++) {
      for (j = 0; j < NUM_PLANETAS; j++)
         if (strcmp(argv[i], planetas[j]) == 0) {
            k = j;
            j = NUM_PLANETAS;
         }
      if (j == NUM_PLANETAS + 1)
         printf("%s é o planeta %d\n", argv[i], k);
      else
         printf("%s não é um planeta\n", argv[i]);
   }
   return 0;
}
```

Se o programa 14.1 tem o nome `planetas.c` e seu executável correspondente tem nome `planetas`, então podemos executar esse programa com uma seqüência de cadeias de caracteres, como mostramos no exemplo abaixo:

```
prompt$ ./planetas Jupiter venus Terra Joaquim
```

O resultado dessa execução é dado a seguir:

```
Jupiter é o planeta 5
venus não é um planeta
Terra é o planeta 3
Joaquim não é um planeta
```

Exercícios

14.1 As chamadas de funções abaixo supostamente escrevem um caractere de mudança de linha na saída, mas algumas delas estão erradas. Identifique quais chamadas não funcionam e explique o porquê.

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

(b) `printf("%c", "\n");`

(c) `printf("%s", '\n');`

(d) `printf("%s", "\n");`

(e) `printf('\n');`

(f) `printf("\n");`

(g) `putchar('\n');`

(h) `putchar("\n");`

14.2 Suponha que declaramos um ponteiro p como abaixo:

```
char *p = "abc";
```

Quais das chamadas abaixo estão corretas? Mostre a saída produzida por cada chamada correta e explique por que a(s) outra(s) não está(ão) correta(s).

(a) `putchar(p);`

(b) `putchar(*p);`

(c) `printf("%s", p);`

(d) `printf("%s", *p);`

14.3 Suponha que declaramos as seguintes variáveis:

```
char s[MAX+1];
int i, j;
```

Suponha também que a seguinte chamada foi executada:

```
scanf("%d%s%d", &i, s, &j);
```

Se o(a) usuário(a) digita a seguinte entrada:

```
12abc34 56def78
```

quais serão os valores de i, j e s depois dessa chamada?

14.4 A função abaixo supostamente cria uma cópia idêntica de uma cadeia de caracteres. O que há de errado com a função?

```
char *duplica(const char *p)
{
   char *q;

   strcpy(q, p);

   return q;
}
```

14.5 O que imprime na saída o programa abaixo?

```
#include <stdio.h>

int main(void)
{
   char s[] = "Dvmuvsb", *p;

   for (p = s; *p; p++)
      --*p;
   printf("%s\n", s);

   return 0;
}
```

14.6 (a) Escreva uma função com a seguinte interface:

```
void maiuscula(char cadeia[])
```

que receba uma cadeia de caracteres (terminada com um caractere nulo) contendo caracteres arbitrários e substitua os caracteres que são letras minúsculas nessa cadeia por letras maiúsculas. Use `cadeia` apenas como vetor, juntamente com os índices necessários.

(b) Escreva uma função com a seguinte interface:

```
void maiuscula(char *cadeia)
```

que receba uma cadeia de caracteres (terminada com um caractere nulo) contendo caracteres arbitrários e substitua os caracteres que são letras minúsculas nessa cadeia por letras maiúsculas. Use apenas ponteiros e aritmética com ponteiros.

14.7 (a) Escreva uma função que receba uma cadeia de caracteres e devolva o número total de caracteres que ela possui.

(b) Escreva uma função que receba uma cadeia de caracteres e devolva o número de vogais que ela possui.

(c) Escreva uma função que receba uma cadeia de caracteres e devolva o número de consoantes que ela possui.

(d) Escreva um programa que receba diversas cadeias de caracteres e faça a média do número de vogais, de consoantes e de símbolos de pontuação que elas possuem.

Use apenas ponteiros nas funções em (a), (b) e (c).

14.8 Escreva um programa que encontra a maior e a menor palavra de uma seqüência de palavras informadas pelo(a) usuário(a). O programa deve terminar se uma palavra de quatro letras for fornecida na entrada. Considere que nenhuma palavra tem mais que 20 letras.

Um exemplo de entrada e saída do programa pode ser assim visualizado:

```
Informe uma palavra: laranja
Informe uma palavra: melao
Informe uma palavra: tomate
Informe uma palavra: cereja
Informe uma palavra: uva
Informe uma palavra: banana
Informe uma palavra: maca

Maior palavra: laranja
Menor Palavra: uva
```

14.9 Escreva um programa com nome `reverso.c` que mostra os argumentos da linha de comandos em ordem inversa. Por exemplo, executando o programa da seguinte forma:

```
prompt$ ./reverso garfo e faca
```

deve produzir a seguinte saída:

```
faca e garfo
```

14.10 Escreva um programa com nome `soma.c` que soma todos os argumentos informados na linha de comandos, considerando que todos eles são números inteiros. Por exemplo, executando o programa da seguinte forma:

```
prompt$ ./soma 81 25 2
```

deve produzir a seguinte saída:

```
108
```