

# Ponteiros

## Aula 07

Ivone P. Matsuno Yugoshi    Ronaldo Fiorilo dos Santos

`ivone.matsuno@ufms.br`    `ronaldo.santos@ufms.br`

Universidade Federal de Mato Grosso do Sul  
Câmpus de Três Lagoas  
Bacharelado em Sistemas de Informação

Algoritmos e Programação II

# Alocação dinâmica de memória

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    int i, j, m, n, **matriz, **pt;
    scanf("%d%d", &m, &n);
    matriz = (int **) malloc(m * sizeof(int *));
    if (matriz == NULL)
        return 0;
    for (pt = matriz, i = 0; i < m; i++, pt++) {
        *pt = (int *) malloc(n * sizeof(int));
        if (*pt == NULL)
            return 0; }
    for (i = 0; i < m; i++)
        for (j = 0; j < n; j++)
            scanf("%d", &matriz[i][j]);
    pt = matriz;
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++)
            printf("%d ", *(pt+i+j));
        printf("\n"); }
    return 0;
}
```

# Alocação dinâmica de memória

- ▶ Além da função `malloc` existem duas outras funções para alocação de memória na `stdlib.h`: `calloc` e `realloc`
- ▶ As três funções solicitam blocos de memória de um espaço de armazenamento conhecido também como *heap* ou ainda **lista de espaços disponíveis**
- ▶ A chamada frequente dessas funções pode exaurir o *heap* do sistema, fazendo com que essas funções devolvam um ponteiro nulo
- ▶ Pior ainda, um programa pode alocar blocos de memória e perdê-los de algum modo, gastando espaço desnecessário

```
p = malloc(...);  
q = malloc(...);  
p = q;
```

# Alocação dinâmica de memória

- ▶ A função **free** é usada para ajudar a resolver o problema de geração de lixo na memória durante a execução de programas:

```
void free(void *pt)
```

- ▶ A função **free** recebe um ponteiro para um bloco de memória que não mais necessitamos:

```
p = malloc(...);  
q = malloc(...);  
free(p);  
p = q;
```

- ▶ A função **free** devolve o bloco de memória apontado por  $p$  para o *heap*
- ▶ O argumento da função **free** deve ser um ponteiro que foi previamente devolvido por uma função de alocação de memória

# Literais e ponteiros

- ▶ Uma **literal** é uma sequência de caracteres envolvida por aspas duplas
- ▶ Exemplo:

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

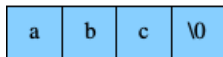
frase de Bill Gates concedendo entrevista à revista *Computer Magazine*

- ▶ Literais ocorrem com frequência na chamada das funções **printf** e **scanf**
- ▶ Quando chamamos uma dessas funções e fornecemos uma literal como argumento, o que de fato estamos passando?

- ▶ 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 sequência de caracteres `\0`

# Literais e ponteiros

- ▶ Por exemplo, a literal `"abc"` é armazenada como uma cadeia de quatro caracteres:



- ▶ Literais podem ser vazias, ou seja, a cadeia `" "` 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 *`
- ▶ `printf` e `scanf`, por exemplo, esperam um valor do tipo `char *` como primeiro argumento

- ▶ Por exemplo:

```
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

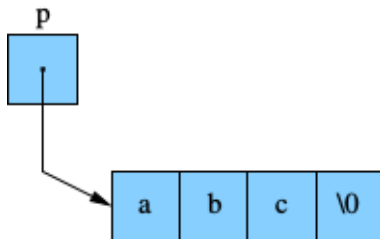
- ▶ Podemos usar uma literal sempre que a linguagem C permita o uso de um ponteiro do tipo `char *`
- ▶ Por exemplo, uma literal pode ocorrer do lado direito de uma atribuição:

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



# Literais e ponteiros

- ▶ Essa atribuição não copia os caracteres de `"abc"`, mas faz o ponteiro `p` apontar para o primeiro caractere da literal



- ▶ Não é permitido alterar uma literal durante a execução de um programa

- ▶ Podemos inicializar uma cadeia de caracteres no momento de sua declaração:

```
char data[13] = "9 de outubro";
```

- ▶ O compilador coloca os caracteres de "9 de outubro" no vetor `data` e adiciona o caractere nulo ao final para que `data` possa ser usada como uma cadeia de caracteres

- ▶ Apesar de `"9 de outubro"` se parecer com uma literal, a linguagem C de fato a vê como uma abreviação para um inicializador de um vetor:

```
char data[13] = {'9',' ','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
  - ▶ é sempre importante garantir que o inicializador tenha menor comprimento que o tamanho do vetor declarado
- ▶ 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

- ▶ declaração um vetor **data** :

```
char data[] = "9 de outubro";
```

- ▶ declaração de um ponteiro **data** para uma literal:

```
char *data = "9 de outubro";
```

- ▶ devido à relação estrita entre vetores e ponteiros, podemos usar as duas versões da declaração de **data** como uma cadeia de caracteres
- ▶ 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

- ▶ devemos ter **cuidado** para não cometer o erro de acreditar que as duas versões da declaração de **data** são equivalentes e intercambiáveis:
  - ▶ 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:

```
char *p;
```

faz com que o compilador reserve espaço suficiente para uma variável ponteiro

- ▶ 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:

```
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

# 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

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

- ▶ observe que omitimos o número de linhas da matriz, que é fornecido pelo inicializador, mas a linguagem C exige que o número de colunas seja especificado



# Vetores de cadeias de 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

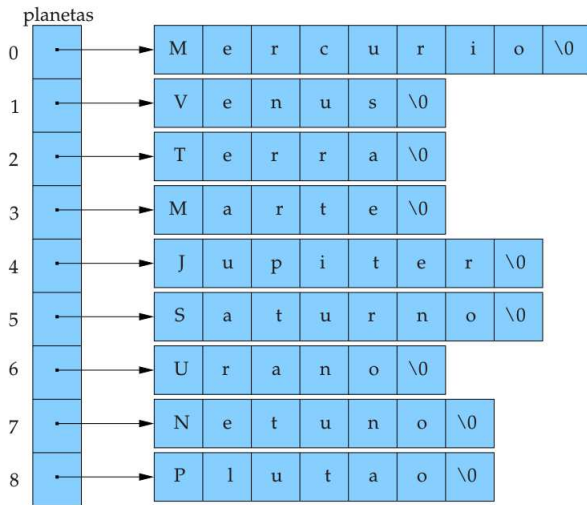
# Vetores de cadeias de caracteres

- ▶ 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 forma de sanar esse problema é usar um vetor cujos elementos são ponteiros para cadeias de caracteres:

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

- ▶ 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
- ▶ o efeito dessa declaração na memória é muito diferente

# Vetores de cadeias de caracteres



# Vetores de cadeias de caracteres

- ▶ 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
- ▶ para acessar um dos nomes dos planetas precisamos 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:

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

# 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:

```
prompt$ ls
```

```
prompt$ ls -l
```

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

- ▶ informações em linha de comando estão disponíveis para todos os programas, não apenas para comandos do sistema operacional

# Argumentos na linha de comandos

- ▶ 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 costumemente têm identificadores **argc** e **argv**

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

- ▶ **argc**, abreviação de “contador de argumentos”, é o número de argumentos de linha de comando, incluindo também o nome do programa
- ▶ **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

# Argumentos na linha de comandos

- ▶ `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 alguém digita:

```
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

# Argumentos na linha de comandos

- ▶ **argv** é um vetor de ponteiros e, por isso, o acesso aos argumentos da linha de comandos é dado da seguinte forma:

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



# Argumentos na linha de comandos

```
#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;
}
```

# Argumentos na linha de comandos

- ▶ supondo que o nome do programa fonte seja `planetas.c` e seu executável correspondente tem nome `planetas`, podemos executar esse programa com uma sequência de cadeias de caracteres, como mostramos no exemplo abaixo:

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

- ▶ resultado dessa execução:

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

1. Dados dois vetores  $x$  e  $y$ , ambos com  $n$  elementos,  $1 \leq n \leq 100$ , determinar o produto escalar desses vetores. Use alocação dinâmica de memória.
2. Dada uma cadeia de caracteres com no máximo 100 caracteres, contar a quantidade de letras minúsculas, letras maiúsculas, dígitos, espaços e símbolos de pontuação que essa cadeia possui. Use alocação dinâmica de memória.
3. Dada uma matriz de números reais  $A$  com  $m$  linhas e  $n$  colunas,  $1 \leq m, n \leq 100$ , e um vetor de números reais  $v$  com  $n$  elementos, determinar o produto de  $A$  por  $v$ . Use alocação dinâmica de memória.

4. Dizemos que uma matriz quadrada de números inteiros distintos é um **quadrado mágico** se a soma dos elementos de cada linha, a soma dos elementos de cada coluna e a soma dos elementos da diagonal principal e secundária são todas iguais.

*A matriz*

$$\begin{pmatrix} 8 & 0 & 7 \\ 4 & 5 & 6 \\ 3 & 10 & 2 \end{pmatrix}$$

*é um quadrado mágico.*

Exemplo:

Dada uma matriz quadrada de números inteiros  $A_{n \times n}$ , com  $1 \leq n \leq 100$ , verificar se  $A$  é um quadrado mágico. Use alocação dinâmica de memória.

5. 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;
}
```

## 6. 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;
}
```

7. (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.

8. (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).



9. 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
```

10. 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
```

11. 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
```