





Programação II

Vetor de Tipo Estruturado e Vetor de Ponteiros


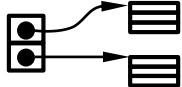
Bruno Feijó
Dept. de Informática, PUC-Rio

Vetores e Tipos Estruturados

- Dado um tipo estruturado T (com 3 campos, por exemplo), podemos ter um vetor de tipos estruturados:

Estrutura T  e vetor de estruturas: $T \text{ } v[2];$ v 

- ou um vetor de ponteiros para o tipo estruturado T :

Estrutura T  e vetor de ponteiros para estruturas: $T * v[2];$ v 

- Note que, neste último caso, o que está guardado em $v[0]$ e $v[1]$ são endereços de uma estrutura (e não a própria estrutura).
- Um exemplo do primeiro caso, com alocação estática de memória:

```
typedef struct ponto Ponto;  
struct ponto  
{  
    float x;  
    float y;  
};
```

```
int main(void)  
{  
    Ponto v[3]; // vetor de struct estatico  
    v[0].x = 0.0f; v[0].y = 0.0f;  
    v[1].x = 10.0f; v[1].y = 0.0f;  
    v[2].x = 5.0f; v[2].y = 10.0f;  
    ...  
    return 0;  
}
```

Vetores e Tipos Estruturados (continuação)

- Um exemplo de vetor de ponteiros para estruturas, com alocação estática de memória:

```
typedef struct ponto Ponto;  
struct ponto  
{  
    float x;  
    float y;  
};
```

```
int main(void)  
{  
    Ponto * v[3]; // alocação estática  
    Ponto p1, p2, p3;  
    p1.x = 0.0f; p1.y = 0.0f;  
    p2.x = 10.0f; p2.y = 0.0f;  
    p3.x = 5.0f; p3.y = 10.0f;  
    v[0] = &p1;  
    v[1] = &p2;  
    v[2] = &p3;  
    ...  
}
```

Mas há uma grande vantagem em usar um array de ponteiros para estrutura ao invés de um array de estruturas: a memória contígua é menor (apenas ponteiros) e as estruturas podem ficar espalhadas na memória.

Note que não há como retornar um ponteiro para um *Ponto* que é uma variável local dentro da *myFunction*, porque a variável deixa de existir quando a função acaba (i.e. estaríamos retornando um valor de endereço inválido). Use malloc!

- Quando passamos um vetor de estruturas ou um vetor de ponteiros para estruturas como argumento de uma função, estamos passando apenas o endereço do início do vetor. Ou seja: a situação é equivalente, em termos de custo.
- Mas, se o retorno é a estrutura ou o ponteiro para a estrutura, há uma diferença importante. O custo é muito alto no primeiro caso, porque a cópia do valor de retorno lida com uma grande área de memória. E.g.:
 - caso vetor de struct: `Ponto myFunction(Ponto * v, int n)`
 - caso vetor de ponteiros para struct: `Ponto * myFunction(Ponto ** v, int n)`



Vetores bidimensionais e vetores de ponteiros

Array (Vetor) Bidimensional

Alunos, em geral, se confundem com a diferença entre vetores bidimensionais e vetores de ponteiros.

`a[][]` é um vetor bidimensional, por exemplo:

```
char a[][15] = {"Bom Dia.", "Boa Tarde.", "Boa Noite."};
```

com a seguinte armazenagem na memória:

Bom Dia.\0	Boa Tarde.\0	Boa Noite\0	
0	15	30	44

O que é equivalente à matriz:

	0	1	2	3	4	5	6	7	8	...					14
0	<i>B</i>	<i>o</i>	<i>m</i>		<i>D</i>	<i>i</i>	<i>a</i>	.	\0
1	<i>B</i>	<i>o</i>	<i>a</i>		<i>T</i>	<i>a</i>	<i>r</i>	<i>d</i>	<i>e</i>	.	\0
2	<i>B</i>	<i>o</i>	<i>a</i>		<i>N</i>	<i>o</i>	<i>i</i>	<i>t</i>	<i>e</i>	\0

Você pode alterar um determinado caracter ou imprimir todo um elemento:

```
a[1][4] = 'X'; // troca T por X no segundo elemento
printf("%s\n",a[1]); // imprime Boa Xarde.
```

Array (Vetor) **Bidimensional** vs Ponteiros

Aritmética de ponteiro e indexação de array são equivalentes em C, mas ponteiros e nomes de array são diferentes !

Quando o array é de uma única dimensão, as facilidades são muitas e terminam fazendo a gente se esquecer de que array e ponteiro são entidades diferentes e de que o compilador as trata de maneira diferente. Veja o Slide 4 do material "Vetores e Alocação Dinâmica".

Quando temos um ARRAY BIDIMENSIONAL `b` como argumento de uma função `g`, por exemplo:

```
char b[3][15] = {"abc", "a", "cd"};  
g(b, 3);
```

a função `g` deve ser:

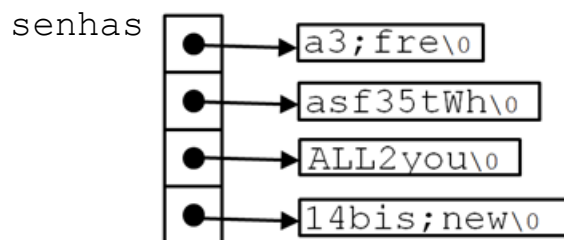
```
void g(char b[][15], int n) {...}  
ou  
void g(char (*b)[15], int n) {...}
```

Ou seja, temos que saber o número de colunas (fazer uma função que independa de saber o número de colunas é complicado !)

Array (Vetor) de Ponteiros

```
char * senhas[] = {"a3;fre", "asf35tWh", "ALL2you", "14bis;new"};
```

onde `senhas` é inicializado com constantes de string: "a3;fre", "asf35tWh", "ALL2you" e "14bis;new", temos a seguinte situação:



Compare com array 2D:

```
char a[][15] = {"a3;fre",  
               "asf35tWh", "ALL2you",  
               "14bis;new"};
```

A vantagem é que cada elemento aponta para strings de tamanhos diferentes. Podemos imprimir um elemento de `senhas`:

```
printf("%s\n", senhas[2]); // imprime ALL2you
```

Mas não podemos alterar um caractere dentro da constante de string (e.g. **NÃO** podemos alterar: `senhas[1][4] = '9';`).

Entretanto, podemos alterar caracteres quando somos nós que alocamos o espaço de memória. Faça um programa que declara estaticamente `senhas` com 2 ponteiros para strings dinâmicos e troque um caractere:

```
char * senhas[2];  
char * s1 = "onlyyou2";  
char * s2 = "justme";  
  
senhas[0] = (char *)malloc((strlen(s1)+1)*sizeof(char));  
strcpy(senhas[0], s1);  
senhas[1] = ...  
...  
*(senhas[0]+7) = '3'; // ou senhas[0][7] = '3';  
printf("%s\n", senhas[0]); // imprime onlyyou3
```

Array (Vetor) vs Ponteiros - Função

Quando a gente não quer desperdiçar memória (como em `char b[3][15] = {"abc", "a", "cd"};` onde "abc" gastou apenas 4 dos 15 que tinha direito) usamos um **VETOR DE PONTEIROS** para strings. Neste caso, o vetor de ponteiros `b` como argumento de uma função `g` poderia ser o seguinte:

```
char * b[] = {"abc", "a", "cd"};
g(b, 3);
```

Neste caso, a função `g` deve ser: **`void g(char ** b, int n) {...}`**

A diferença é que agora temos "ponteiro" e não "nome de array"!

Melhor mesmo é quando a gente tem tudo como ponteiros ("melhor" no sentido de que fica tudo dinâmico e sob total controle nosso):

```
char ** b;
b = (char **)malloc(3*sizeof(char *));
b[0] = (char *)malloc(strlen("abc")+1);
strcpy(b[0], "abc");
b[1] = (char *)malloc(strlen("a")+1);
strcpy(b[1], "abc");
...
g(b, 3);
```

Veja outro exemplo de Totalmente Dinâmico no slide a seguir

E, neste caso, a função `g` é igual à anterior: **`void g(char ** b, int n) {...}`**

Array (vetor) de Ponteiros – Exemplo Totalmente Dinâmico

- Antes vimos duas maneiras para arrays com strings:

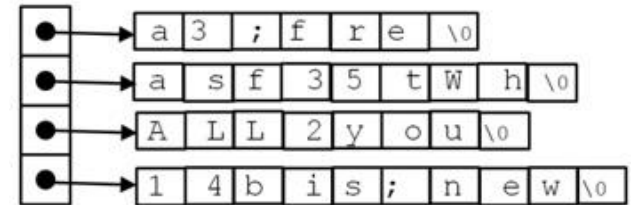
```
char a[][15] = {"a3;fre",  
               "asf35tWh", "ALL2you",  
               "14bis;new"};
```

é array 2D! Não é array de ponteiros!



- Alternativamente, podemos alocar toda a matriz como ponteiros de ponteiro.

```
int id;  
char * temp;  
char ** senhas;  
senhas = (char **)malloc(4*sizeof(char *));  
temp = "a3;fre";  
senhas[0] = (char *)malloc((strlen(temp)+1)*sizeof(char));  
strcpy(senhas[0],temp);  
temp = "asf35tWh";  
senhas[1] = (char *)malloc((strlen(temp)+1)*sizeof(char));  
strcpy(senhas[1],temp);
```



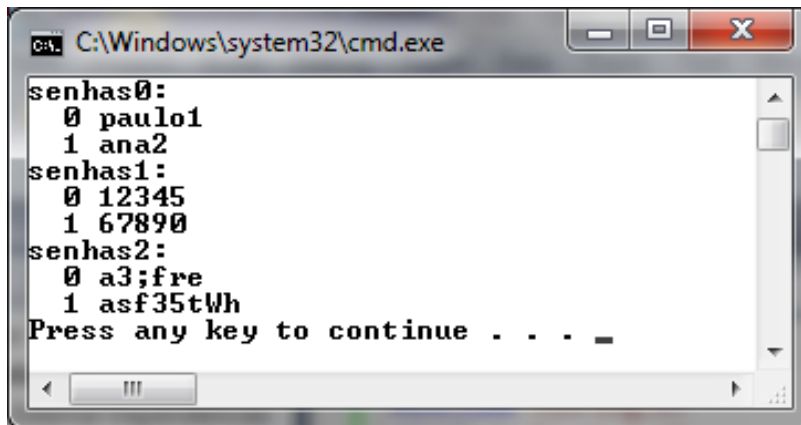
- Os cabeçalhos das funções podem, independente dos dois métodos acima (para arrays de ponteiros), sempre ser: ... *nomeFuncao*(..., char ** senhas, ...)

Exercício

Faça um projeto contendo os arquivos prog1.c, senhas.c e senhas.h que imprime os seguintes arrays: **senhas0** que é um array bidimensional; **senhas1** que é um array de ponteiros alocado estaticamente; e **senhas2** que é um array de ponteiros alocado dinamicamente. Respeite as seguintes condições:

- [1] Para montar o array senhas2, defina variáveis contendo constantes de string (e.g. char * a = "a3;fre"; char * b = "asf35tWh");)
- [2] Faça uma função para cada tipo de array (e.g. printSenha0, printSenha1 e printSenha2).
- [3] No final do programa libere todas as memórias possíveis, sem deixar "órfãos" (i.e. espaços de memória cujas referências são perdidas).

Na figura abaixo está um exemplo de saída para arrays com duas senhas.



```
C:\Windows\system32\cmd.exe

senhas0:
 0 paulo1
 1 ana2
senhas1:
 0 12345
 1 67890
senhas2:
 0 a3;fre
 1 asf35tWh
Press any key to continue . . . _
```

Você pode resolver este exercício sem usar os 3 arquivos (senhas.h, senhas.c e prog1.c). Faça um único arquivo com tudo.

Gabarito do Exercício – Vetores de Ponteiros

prog1.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "senhas.h"

int main(void)
{
    char * a = "a3;fre";
    char * b = "asf35tWh";
    int n = 2;
    char senhas0[][NMAX]={ "paulo1", "ana2" };
    char * senhas1[]={ "12345", "67890" };
    char ** senhas2;

    senhas2 = (char **)malloc(n*sizeof(char *));
    senhas2[0] = (char *)malloc(strlen(a)+1);
    strcpy(senhas2[0],a);
    senhas2[1] = (char *)malloc(strlen(b)+1);
    strcpy(senhas2[1],b);

    printf("senhas0:\n");
    printSenhas0(n,senhas0);
    printf("senhas1:\n");
    printSenhas1(n,senhas1);
    printf("senhas2:\n");
    printSenhas2(n,senhas2);

    free(senhas2[0]);
    free(senhas2[1]);
    free(senhas2);

    return 0;
}
```

senhas.h

```
#define NMAX 15
void printSenhas0(int n, char (*senhas)[NMAX]);
void printSenhas1(int n, char ** senhas);
void printSenhas2(int n, char ** senhas);
```

senhas.c

```
#include <stdio.h>
#include "senhas.h"

void printSenhas0(int n, char
(*senhas)[NMAX])
{
    int id;
    for (id=0; id<n; id++)
        printf("%3d %s\n",id,senhas[id]);
}

void printSenhas1(int n, char ** senhas)
{
    int id;
    for (id=0; id<n; id++)
        printf("%3d %s\n",id,senhas[id]);
}

void printSenhas2(int n, char ** senhas)
{
    int id;
    for (id=0; id<n; id++)
        printf("%3d %s\n",id,senhas[id]);
}
```