



Guião aula 4

Guião da Ficha de Trabalho 4

Introdução aos Arrays

Os arrays encontram-se entre as estruturas de dados mais antigas e importantes em programação.

Em C, arrays são uma fonte muito comum de erros ou falhas de segurança. Isto deve-se ao facto de estarem relacionados com o conceito de apontador de memória de elementos, que se encontram armazenados num espaço contíguo em memória.

Por exemplo, o array:

```
int arr[5] = {10, 2, 13, 64, 7};
```

Encontra-se representado em memória da seguinte forma:

Endereço	Valor	Índice
0x3bf4fbfd28	10	0
0x3bf4fbfd2c	2	1
0x3bf4fbfd30	13	2
0x3bf4fbfd34	64	3
0x3bf4fbfd38	7	4

Como tal, é possível aceder a um elemento de um array de tamanho N pelo seu índice, ou seja, a sua posição no array. Este índice começa em 0 e termina em N-1.

Qualquer variável declarada como array é, na verdade, **um apontador para o primeiro elemento do array**. Ou seja, a seguinte expressão é verdadeira (onde 8 é o endereço de memória do array):

```
arr == &arr[0]
```

Isto também permite o uso de **aritmética de apontadores**, como por exemplo (onde * é o operador de desreferência):

```
arr[2] == *(arr + 2);
```

Isto nota-se também no uso de **char* para representar strings**, que não são mais do que arrays de caracteres, e como tal, equivalentes a um apontador para o primeiro elemento do array:

```
char *hello = "hello, world";
```

É equivalente a

```
char hello[] = { 'h', 'e', 'l', 'l', 'o', ',', ' ', 'w', 'o', 'r', 'l', 'd' };
```

Este acesso direto à memória encontra-se mais presente em linguagens de mais baixo nível como C, ao contrário de linguagens como Java ou C#, onde a gestão da memória é feita automaticamente, o que confere uma grande vantagem de eficiência a linguagens da família. Contudo, pode levar a riscos de *segmentation fault*, *stack overflow*, *heap corruption*, entre outros.

Nota: no final deste guião, são introduzidos com mais detalhe os conceitos essenciais de arrays em C.

Exercícios propostos para praticar

- Escreva um programa em C que seja capaz de efetuar a leitura de cinco números inteiros para um array de 5 posições e de seguida os mostre pela ordem em que foram introduzidos.
- Escreva um programa em C que, dado um array X de inteiros de N posições, transfere para um segundo array Y apenas os valores superiores à média dos elementos de X.
Ex: dado X = {1, 8, 2, 5, 5}, deve colocar em Y os valores {8, 5, 5} (porque a média é 4.2).
- Desenvolva uma **função** em C que dadas duas strings verifica se a primeira contém a segunda. Caso contenha, devolve o índice da 1ª string onde o conteúdo da segunda string começa.

Tarefas de avaliação

Pretende-se que implemente em C as funções com os seguintes protótipos:

```
void soma_elemento(int *arr, int dim, int idx);  
void roda_esq(int *arr, int dim, int shifter);  
int remove_menores(int *arr, int dim, int valor);
```

- Desenvolva uma **função** em C que recebe um array de inteiros, a sua dimensão e um índice qualquer (menor que a dimensão). A função deverá alterar o valor do elemento no índice indicado por um valor dado. Ex: dado {1, 5, 4, 3, 2}, a dimensão 5 e o índice 2, deve devolver: {5, 9, 8, 7, 6}, que corresponde a {1+4, 5+4, 4+4, 3+4, 2+4}.
- Desenvolva uma **função** em C que recebe um array, a sua dimensão e ainda um *shifter* (qualquer inteiro maior ou igual a 0). A função deve retornar o array com os elementos deslocados para a esquerda pelo shifter.
Ex: dado o array {1, 2, 3, 4, 5}, a dimensão 5 e o shifter 3, deve devolver: {4, 5, 1, 2, 3}. Ex: dado o array {1, 2, 3, 4, 5}, a dimensão 5 e o shifter 7, deve devolver: {3, 4, 5, 1, 2}.
- Desenvolva uma **função** em C que dado um array, a sua dimensão e um valor, remova desse array os elementos menores do que o valor (movendo-os para a direita) e mantenha a sua ordem. Ex: dado o array {3, 7, 2, 1, 4}, a dimensão 5 e o valor 4, deve devolver: {7, 4, 3, 2, 1}.
- Faça um programa principal que leia o número da tarefa, a dimensão do array, os seus elementos e o último argumento e que imprima o array após invocar a função correspondente.

Conceitos Essenciais de Arrays em C

- **Declaração de Arrays.** Um array em C é declarado, tal como uma variável normal, com o tipo e o nome. É adicionada a capacidade do array, escrevendo retos - [e].
 - `int a[5];`
 - `int a[MAX];` (usando `#define MAX 5`)
 - `int a[n];` (se `n` foi previamente declarado)
- **Inicialização de Arrays.** Um array pode ser inicializado com os valores em todas as suas posições de uma única vez. Para isso, usam-se chavetas - { e } - .
 - `int a[] = {12, 1, 32, 4, 6};`
 - `int a[3] = {12, 1, 32};`
 - `int a[10] = {12};` //OK (as outras posições não são inicializadas)
 - `int a[2] = {12, 1, 32, 4};`
 - Neste caso, o compilador vai mostrar um **warning**, pois é declarado um array de 2 elementos e inicializado com 4.

Notas:

- Quando se declara e inicializa um array (ao mesmo tempo), não é necessário indicar o seu tamanho (o compilador tratará disso).
- Nada impede declarar um array numa instrução e iniciá-lo noutra a seguir.
- **Escrita em Arrays.** Para atribuir um valor a uma posição no > array, é necessário especificar o array e o índice da posição na > qual queremos escrever.
 - `a[0] = 25;`
 - `a[10] = 11;`
- **Leitura em Arrays.** Para ler um valor num índice específico é necessário especificar o array e qual o índice que queremos ler (novamente, os índices dos
 - `int x = a[4];`

Conceitos Essenciais de Memória em C (foco em arrays)

- **Disposição em memória.** A memória pode ser vista como um bloco. Quando um array é declarado, o compilador aloca a memória necessária para guardá-lo.
 - E.g., quando o array é declarado como sendo array de 5 inteiros, então $5 \times 4 = 20$ bytes são alocados na memória.
- **Teoria de Apontadores (simplificada).** A memória pode ser vista como uma cidade. Cada bloco de memória tem um endereço único real, e nesse endereço um inquilino, e portanto sabendo o nome da casa, saberemos quem a habita. Neste contexto, os arrays são vistos como prédios com um único apartamento por andar (e portanto um único inquilino por andar). Assim, o primeiro inquilino mora no piso 0, o segundo mora no piso 1, etc.
 - Por exemplo, vejamos o array `a` de 5 posições. Se queremos saber quem mora no piso `N`, basta verificar o endereço do prédio `a` e subir ao piso em `a[N]`.
 - `int a[5];`

Declara um array de 5 inteiros (um prédio de 5 apartamentos) com um endereço único `a`. Assim, `a+0=a` é o endereço para o primeiro elemento do array, e

O governo da cidade só conhece os endereços reais, e não o nome das casas nem quem as habita. Portanto, para saber quem vive onde, e onde alojar alguém, o governo precisa de um sistema de apontadores. `&n` fornece ao governo o endereço real da casa `n`. `*y` fornece ao governo o habitante que habita no endereço real `y`.

Para facilitar alguns processos, o governo usa apontadores, que não são mais do que nomes mais bonitos que apontam para endereços reais. Todos os arrays são vistos como apontadores.

- `a == &a` SIM Se `a` for um array ou um apontador.
- `*a == a[0]` SIM Se `a` for um array ou um apontador.
- `a[i] == *(a+i)` SIM
- `n == *(&n)` SIM
- `int *p` o governo criou um apontador `p` para um endereço qualquer.
- `p = x` o apontador `p` aponta agora para o endereço `x`
- `*p == *x` SIM, o habitante no endereço `p` e `x` é o mesmo!
- **Array argumento de Função.** Até agora, argumentos de funções são passados por **valor**, isto é, os argumentos de uma função são cópias dos valores das variáveis em memória.

No entanto, os arrays podem ser grandes e copiar um array pode ser complexo. Assim, arrays são sempre passados por **referência**. Passar um argumento por referência significa que a função pode alterar o conteúdo do array original.

Há várias formas possíveis de declarar uma função que tem um array como parâmetro:

```
int func(int a[10]);
int func(int *a);
int func(int a[]);
int func(int a[], int size);
int func(int *a, int size);
```

As duas últimas formas são preferíveis quando foi necessário percorrer o array: o endereço do array é um parâmetro e o seu tamanho é outro. As outras formas permitem ao programador saber quando acaba o array, como por exemplo no caso das strings que acabam com `'\0'`.

alguma forma de saber quando acaba, como por exemplo no caso dos arrays que acabam com `\0`.

Funções que devolvem arrays

```
// Cenário 1
int *func(...) {
    int a[10];
    ...
    return a;
}

// Cenário 2
int *func(...) {
    int *a = (int *) malloc(tamanho * sizeof(int));
    ...
    return a;
}

// Cenário 3
void func(int a[], int size) {
    ...
}
```

O primeiro cenário é errado! O compilador de C irá gerar um **warning** porque um endereço de algo declarado dentro de uma função está a ser retornado. Isto não é permitido. Logo que sairmos da função, o endereço deixa de estar protegido e pode ser reutilizado por outras funções!

O segundo cenário implica alocar primeiro o espaço usando **malloc** e depois devolve o array que foi alocado.

No terceiro cenário, a função não retorna nada, mas como o array foi passado por referência, ele continua a existir assim que a função termina (ao contrário do primeiro cenário, onde a alteração dentro da função seria visível fora da função).