Ponteiros

Ponteiros e Vetores

Suponha a seguinte declaração:

int v[3];

Ao encontrar essa linha, o compilador irá proceder do seguinte modo:

1. Alocará na memória RAM um espaço de memória correspondente a 3 elementos do tipo "int". No caso, 3 x 4 = 12 bytes.
2. Fará com que a palavra "v" seja o nome pelo qual se pode modificar e alterar os valores armazenados nesse espaço de memória de 12 bytes.

Esquematicamente, supondo que esse espaço de memória alocado para "v" seja o endereço 1000:

Endereço	RAM
1000	Lixo
1004	Lixo
1008	Lixo

v é como se fosse sinônimo do número 1000.

A declaração de um vetor em C, essencialmente, diz que o nome "v" é o endereço de memória da 1ª posição do vetor na memória, ou equivalentemente, é o endereço do 1° elemento do vetor.

Se escrevêssemos a seguinte linha **printf("%p", v);** seria impresso na tela o valor 1000, pois p é sinônimo para o valor 1000.

E, como já vimos que para armazenar um valor em um endereço temos que usar o operador unário "*", então para armazenar um valor no endereço v, temos que escrever a linha:
*v=1. Esquematicamente, ficaria o seguinte:

Endereço	RAM
1000	1
1004	Lixo
1008	Lixo

E se quiséssemos armazenar um outro valor no vetor v? Simples, basta incrementar o endereço v e armazenar o valor nesse novo endereço por meio do operador unário "*": *(v+1) = 2. Esquematicamente ficaria do seguinte modo a memória:

Endereço	RAM
1000	1
1004	2
1008	Lixo

Notar que ao somar 1 ao endereço **v**, é incrementado de um valor correspondente ao tamanho do tipo do ponteiro **v** (no caso, é feita a operação 1000 + 4, pois v é um ponteiro para um inteiro que ocupa 4 bytes na memória), pois essa é a nova posição de um elemento apontado por um ponteiro do tipo **v**. **Logo, o TIPO BASE de um ponteiro é importante!**

Em C, também é possível acessar os elementos de um vetor por meio de um índice, do seguinte modo:

```
v[2]= 3;
```

Esquematicamente, a memória ficaria assim:

Endereço	RAM
1000	1
1004	2
1008	3

Concluindo, essa forma v[2]=3 é equivalente a escrever *(v + 2)=3. Só que há uma diferença, é muito mais rápido acessar o vetor por meio de um ponteiro para o mesmo do que por meio da indexação via colchetes. Veja a comparação entre o tempo de execução dos dois códigos seguintes como exemplo:

```
int v[] = {1, 2 , 3}, i;
for (i = 0; i < 3; i++) printf("%i", v[i]);
```

(1000); (1000 + 1 * 4); (1000 + 2 * 4)

```
int v[] = {1, 2 , 3}, i;
int *p = v;
for (i=0; i < 3; i++) printf("%i", *p++);
```

(1000); (1000+4); (1004+4)

Como a operação de multiplicação demora mais tempo para ser calculada do que a operação de adição simples, o laço que usa ponteiros será muito mais rápido de ser executado.

Continuando, se o nome v nada mais é do que um sinônimo para o endereço do 1° elemento na memória então, eu posso armazenar esse endereço em uma variável do tipo ponteiro, correto? Sim.

Veja o seguinte trecho de código:

```
int v[3] = {1, 2, 3};
int *p;
p = v;
```

Na primeira linha aloco um espaço de memória correspondente a 3 números inteiros, definindo o nome v como sendo correspondente ao endereço do primeiro número inteiro. Esquematicamente, temos:

Endereço	RAM
1000	1
1004	2
1008	3

Na segunda linha, aloco mais 4 bytes na memória para o ponteiro p. Esquematicamente, teríamos:

Endereço	RAM
1000	1
1004	2
1008	3
1012	1000

Na terceira linha, armazeno em p o valor 1000, pois o nome v é um "sinônimo" para esse valor.

Logo, posso varrer v, por meio do ponteiro p, igual ao modo como eu varreria o vetor por meio do nome v. Ex.: **for (i = 0; i < 3; i++) printf("%d", *(p+i));**

Veja na tabela a seguir todas as combinações possíveis de impressão:

Linha	Resultado	Explicação
printf("%p ", v)	1000	Endereço do 1° elemento de v
printf("%p ", &v[0])	1000	Endereço do 1° elemento de v
printf("%i ", *v)	1	Valor armazenado no endereço 1000
printf("%p ", &v)	1000	Equivalente a &v[0]
printf("%i ", v[0])	1	Valor armazenado no endereço 1000
printf("%p ", p)	1000	Valor armazenado no endereço 1012
printf("%i ", *p)	1	Valor armazenado no endereço 1000
printf("%p ", &p)	1012	Endereço da variável p na memória

Mais um detalhe, um vetor **v** de inteiros pode ser declarado dos dois modos:

```
int v[3] = {1, 2, 3};
ou
int v[] = {1, 2, 3};
```

No primeiro modo você declara explicitamente quantos elementos vai ter o vetor, conseqüentemente, o compilador vai verificar se você não atribuiu mais valores do que a capacidade declarada para o vetor.

No segundo modo você pode inicializar com quantos elementos você quiser o vetor, pois o compilador se encarregará de alocar espaço para todos os elementos.

Como última observação importante, você não pode incrementar ou decrementar o próprio nome v, pois como já vimos v deve ser entendido como um sinônimo fixo do 1° endereço de memória do espaço alocado para os 3 números inteiros. Portanto, códigos do tipo seguintes são ilegais:

```
int v[3] = {1, 2, 3};
int *pi;
v++; /* equivalente a v = v + 1, que é ilegal, pois não é permitido modificar o valor de v */
v = pi; /* ilegal. Não é possível alterar o valor do nome v */
```

Notas sobre strings

Todo esse raciocínio construído para um vetor de inteiros também é válido para um vetor de caracteres, a nossa tão conhecida string!

Mas uma diferença interessante deve ser ressaltada, quando se trata de um vetor de caracteres, é possível fazer 3 tipos de inicializações ao se declarar uma string.

char v[3] = "ok"; /* lembre-se que uma string tem que ter espaço para armazenar um '\0' */ ou
char c[] = "ok"; /* o próprio compilador vai se encarregar de alocar espaço suficiente */
ou
char *c = "ok"; /*foi alocado 4 bytes de memória para o ponteiro c */

Os dois primeiros modos são equivalentes as declarações já vistas para vetores de inteiros, já o terceiro não, portanto, vamos nos deter com mais detalhe nesse terceiro tipo de declaração.

Primeiro ponto, não é possível declarar um vetor de inteiros de um modo semelhante: int *v = {1, 2, 3};

Essa construção não é valida em C!

Por que então é válida quando se trata de caracteres? Vamos ver o porquê.

Primeiro ponto, quando se faz "int *v;" ou "char *v;" a obrigação do C é alocar um espaço de 4 bytes na memória e fazer com que v seja o "sinônimo" para o endereço desse espaço alocado.

Dito isso, ao escrever uma linha "int *v = {1, 2, 3};" o compilador C teria que fazer com que v contivesse o endereço de memória onde estivesse armazenado o valor 1 na memória. Só que o compilador C não aloca espaço para o valor 1, nem o valor 2 e nem o valor 3. Logo, não há como C armazenar na memória esses valores e fazer com que v aponte para o endereço deles.

Já ao escrever uma linha char *v = "ok"; no caso específico de strings constantes (no caso a palavra "ok" é uma string constante porque foi utilizada como valor inicial de um ponteiro para caracter, ao mesmo tempo em que o ponteiro para caractere foi declarado), o compilador C cria internamente (entenda como "aloca espaço") uma tabela e armazena nela essa string constante (palavra "ok") e todas as outras strings constantes que porventura existirem.

Como conseqüência da existência dessa tabela que é válida essa declaração: **char *v="ok";**

Pois, **v** armazena o endereço do primeiro caracter (letra 'o') da string "ok" armazenada nessa tabela interna criada pelo compilador C.

O problema é que, normalmente, os compiladores alocam essa tabela em uma área de memória com permissão somente para leitura, ou seja, não é possível modificar o valor de uma **string** constante, pois como o próprio nome já mostra, ela é **constante**!

Comprove isso, digitando o seguinte trecho de código e verificando o que acontece:

```
char *v = "ok";
v[0] = 'A';
```

Também é importante ressaltar que, como v é um ponteiro, se você andar com ele, vai perder o endereço de onde está armazenada a **string** "ok" na tabela de constantes criada pelo C. Por exemplo:

```
char *v="ok";
char c = 'A';
v = &c;  /* perdi o endereço da memória onde se encontra a string "ok"! */
```

Outros modos de inicialização de um vetor:

```
int v[5] = {[2] = 1};  /* inicializo o 3° elemento de v com o valor 1 */
int v[5] = {[4] = 'A', [2]=1}; /*inicializo o 5° e 3° e elementos de v com os valores 65 e 1, respectivamente*/
int v[] = {[4] = 'A', [2]=1}; /*inicializo o 5° e 3° e elementos de v com os valores 65 e 1, respectivamente*/
char v[] = {'o', 'k'}; /* inicializo v com as letras 'o' e 'k'. Note que falta o '\0' */
char v[2] = {'o', 'k'}; /* inicializo v com as letras 'o' e 'k'. Note que falta o '\0' */
```

Finalizando, algumas possibilidades de manipulação de *strings* constantes:

```
char *s;
s = "Me chamo"
    "...";          /* o compilador concatenará automaticamente as duas strings */
putchar(3["Estranho"]);
```

Analisando:

3["Estranho"] é equivalente a **3 + "Estranho"**, pois a *string* "Estranho" é avaliada como um ponteiro para o local onde essa *string* é armazenada. Continuando, por comutatividade, **3+"Estranho"** é equivalente a **"Estranho" + 3** que é equivalente a **"Estranho"[3]**. Esta última expressão é evidentemente legal e representa o caractere de índice 3 da string **"Estranho"**.

Exercícios:

1. Para cada uma das linhas de código seguintes, diga quantos bytes são alocados na memória:
   a. `int v[] = {1, 2};`
   b. `int v[3] = {1};`
   c. `int v[] = {[6] = 3};`
   d. `char *c = "ok";`
   e. `char c[] = "ok";`
   f. `char c[2] = "o";`
   g. `int n; scanf("%d", &n); int v[n];` **Obs.**: compile usando a opção "-pedantic" neste caso.

   Dica: se tiver dúvidas, use o bom e velho operador **sizeof**!

2. Para cada uma das linhas de código seguintes, diga se a mesma é "legal" (compila 'ok' e tem sentido), "ilegal" (não compila), ou "estranha" (compila 'ok', mas não tem

sentido ou pode gerar erros em tempo de execução) e o porquê.

a. int v[];
b. int v[]={[2]=-1};
c. char *c = 3;
d. char *c = (char *) 3;
e. char *v = “ok”; char *c; char a = ‘A’; c = &a; v = c;
f. char *v = “testando”; char **c; c = &v; **c = ‘O’;
g. int v1[2]; int v2[2]; v2[1] = v1[1];
h. int v1[2] = {0, 1}; int v2[2] = {[1]=-1}; v2[1] = v1[1];
i. int v[1] = {1, 2};
j. int v1[2]; int v2[2]; v2 = v1;