

Cap. VII – PONTEIROS

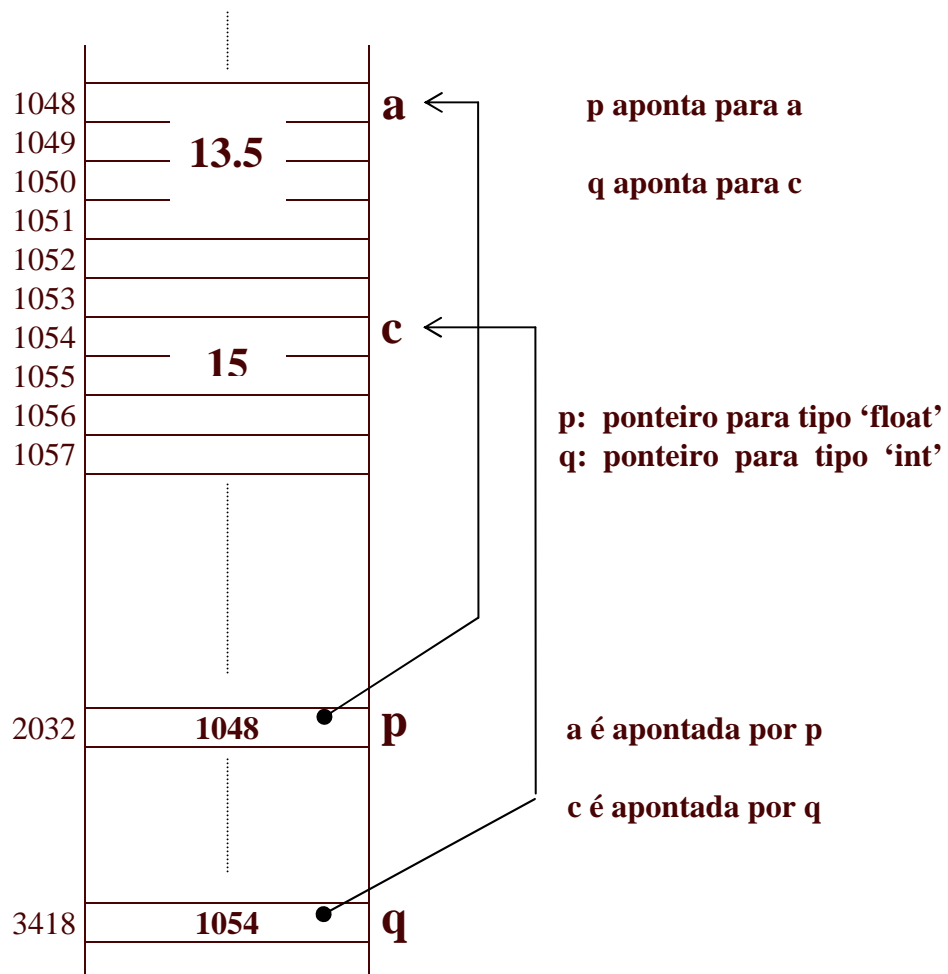
7.1 Declarações e Atribuições

- Até agora, variáveis foram usadas para guardar valores:

float a, b; a 13.5 b 0.7

int c; **char** d; c 15 d ';'

- Variáveis do tipo ponteiro guardam endereços de outras variáveis:



- Declarações:

float *p; int *q;

- Valores possíveis para ponteiros:

– Zero (lugar nenhum) } endereços
 – Inteiros positivos } de memória

- Possíveis atribuições, considerando as declarações anteriores de a, c, p, q:

p = &a; q = &c;
 p = 0; p = NULL; (equivalentes)
 p = (int*) 1776;

- Na última, é necessária uma conversão de tipos:

1776 = inteiro 1776
 (int*) 1776 = endereço de memória 1776
 (memória dividida em blocos de bytes correspondentes a inteiros)

- Conteúdo de um endereço:

Se **p** é um ponteiro,
 ***p** é o valor da variável apontada por **p**.

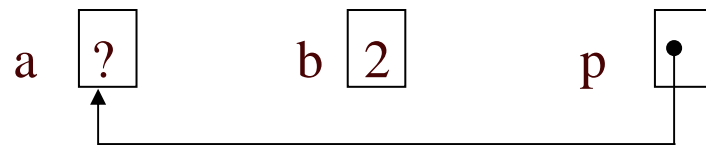
- **Exemplo 7.1:** Sejam as declarações:

int a, b = 2, *p;

- Valores das variáveis:

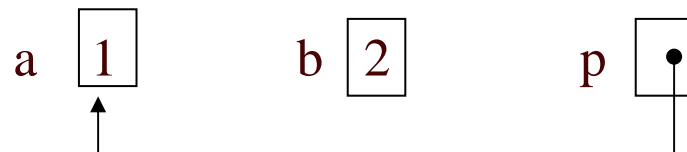


- Com a atribuição $p = \&a$; tem-se



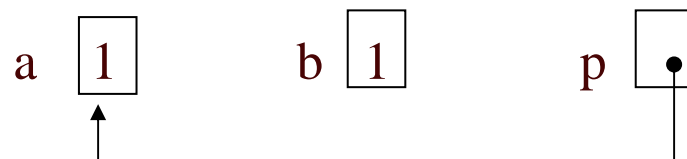
p é definido mas *p é indefinido

- Com a atribuição $*p = 1$; tem-se



$*p = 1$; equivale a $a = 1$;

- Com a atribuição $b = *p$; tem-se



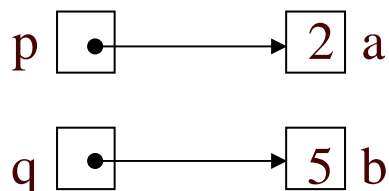
b = *p;
equivale a
b = a;

- **Exemplo 7.2:** sejam as declarações:

int a = 2, b = 5, *p = &a, *q = &b;

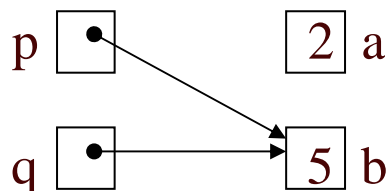
A inicialização é de p e q e não de *p e *q

- Valores das variáveis:



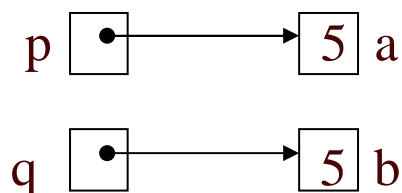
Os valores de *p e *q são 2 e 5, respectivamente.

- Com a atribuição p = q; tem-se



Os valores de *p e *q são ambos 5.

- Voltando à situação inicial, com a atribuição *p = *q; tem-se



- **Exemplo 7.3:** teste de compatibilidade de tipos

Seja o programa:

```
#include <stdio.h>
void main ( ) {
    long a = 2, b = 10000, *p = &a, *q = &b;
    *p = q;
    printf ("a = %ld; b = %ld;\np = %ld; q = %ld;\n*p = %ld; *q = %ld;");
    a, b, p, q, *p, *q);
}
```

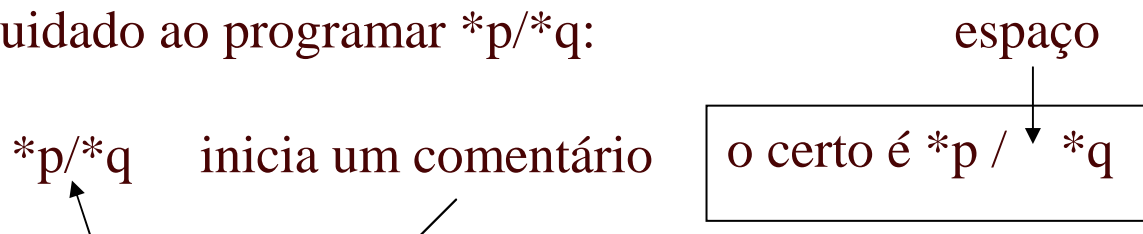
Resultado:

```
a = 580853824;  b = 10000;
p = 580853828;  q = 580853824
*p = 580853824; *q = 10000;
```

Trocando o comando `*p = q` por `p = *q` houve um erro de execução.

`p` recebe o endereço 10000 que é uma área ocupada pelo sistema operacional e proibida para o usuário (programa).

- Cuidado ao programar `*p/*q`:



- **Obs:** A passagem de parâmetros por referência é implementada em C por meio de ponteiros.
- No exemplo 6.10 do Cap.VI:

```
void trocar (int *p, int *q) {  
    int aux;  
    aux = *p; *p = *q; *q = aux;  
}
```

na função **main ()**: trocar (&i, &j);

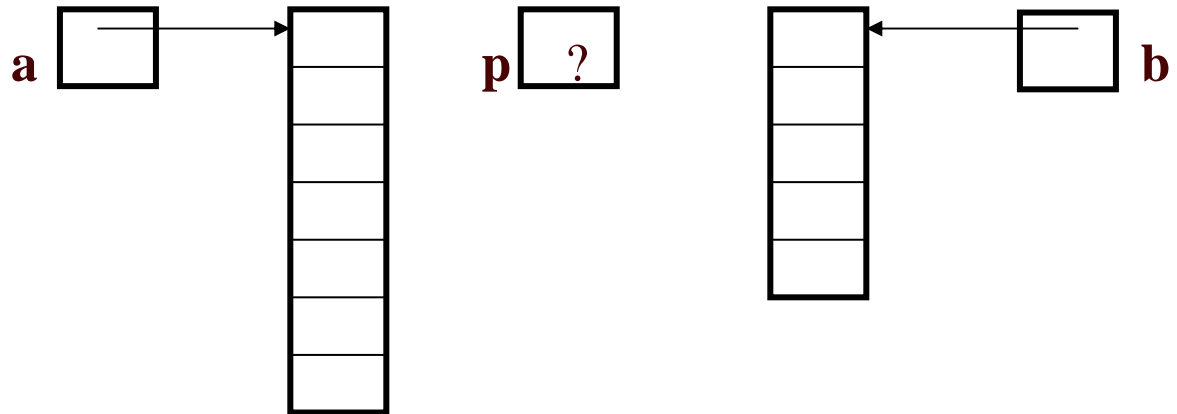
- os endereços de i e j são enviados para os ponteiros p e q.
- dentro da função **trocar**, os valores dos locais apontados por p e q são trocados.

7.2 – Relação entre Variáveis Indexadas e Ponteiros

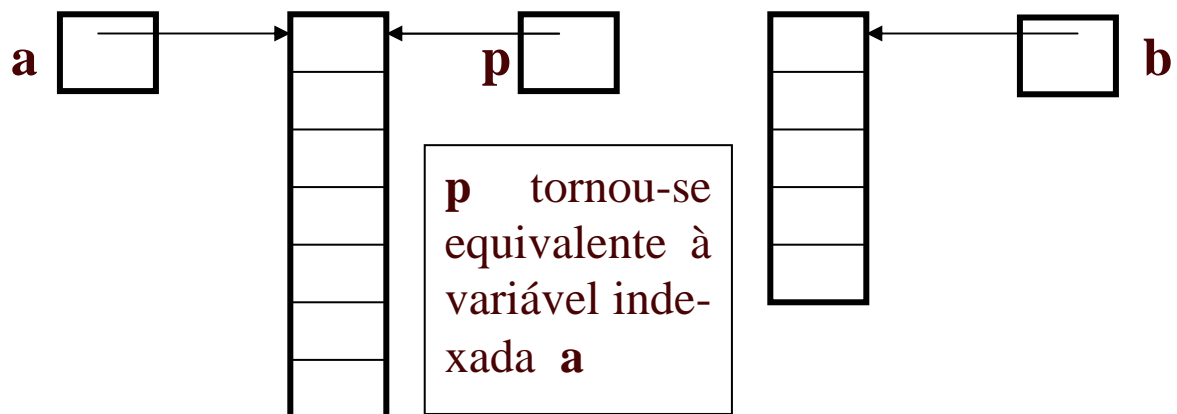
- O nome de uma variável indexada é um endereço. É um ponteiro.
- O valor desse ponteiro é constante durante a execução, não podendo ser alterado por nenhum comando do programa.

- **Exemplo 7.4:** sejam as declarações `int a[7], b[5], *p;`

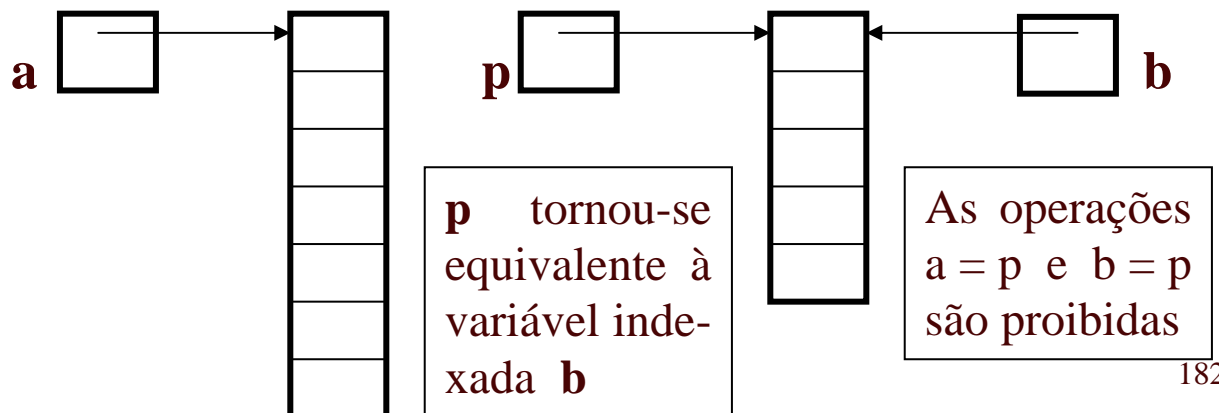
– Situação inicial:



– Situação depois da atribuição `p = a;`



– Situação depois da atribuição `p = b;`



- Ponteiros podem ter subscritos e variáveis indexadas admitem o operador unário ‘*’.

- Supondo as declarações:

int i, a [50], *p;

– a [i] é equivalente a *(a+i).

– *(p+i) é equivalente a p[i].

- A diferença entre ponteiros e variáveis indexadas é :

– nome de uma variável indexada é um ponteiro de valor fixo. O endereço dentro dele não muda. O ponteiro tem valor variável.

– p = &i; é permitido; a = &i; não é permitido.

– a [1] é sempre o mesmo local de memória;
p [1] pode mudar de local.

- Sabe-se que a é o endereço de a[0], logo

p = a ; equivale a p = &a[0];

p = a+1; equivale a p = &a[1];

- Várias maneiras de somar os elementos de **a**:

```
for (soma = 0, p = a; p < &a[50] ; ++p) soma += *p;
for (soma = 0, i = 0; i < 50; ++i) soma += *(a+i);
for (soma = 0, p = a, i = 0; i < 50; ++i) soma += p[i];
```

- Se **p** é um ponteiro para um tipo qualquer, **p+1** é o endereço da próxima variável do mesmo tipo.
- **Exemplo 7.5:** seja o programa

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

```
void main ( ) {
```

```
    double a [5], *p, *q, *r;
```

```
    p = a; q = p + 1; r = p + 2;
```

```
    printf (“q – p = %ld; \nr – p = %ld; \n”, q – p, r – p);
```

```
    printf (“end q – end p = %ld; \n”, (long) q – (long) p);
```

```
    printf (“end r – end p = %ld; \n”, (long) r – (long) p);
```

```
}
```

– **Resultado**

```
q – p = 1;
r – p = 2;
end q – end p = 8;
end r – end p = 16;
```

– **q – p** e **r – p** funcionam como diferenças de índices.

– **(long) q – (long) p = 8**: **p** aponta para uma variável do tipo *double* e **q** aponta para a *double* seguinte; *double* ocupa 8 bytes.

- **Obs:** o uso de ponteiros para o tipo *void*:

- Sejam as declarações

int *p; float *q; void *r;

- Os comandos `p = q;` e `q = p;` são ilegais.

- Caso isso seja necessário, usar o ponteiro para *void* como intermediário, na versão 4.5 do Borland C++:

`p = r = q;` é legal (Versões mais novas não aceitam).

ou então

`p = (int *) q; e q = (float *) p;`

7.3 – Alocação Dinâmica de Memória

- Muitas vezes é útil reservar espaço para uma variável indexada, em tempo de execução.
- Sabendo-se o seu número de elementos, reserva-se espaço necessário e suficiente para essa variável.
- Em primeiro lugar, tal variável deve ser declarada como ponteiro.
- Depois, usa-se a função *malloc* para reservar-lhe espaço em tempo de execução.

- **Exemplo 7.6:** Leitura de vetores

```
#include <stdio.h>
#include <stdlib.h>
void main ( ) {
    int m, i, *a, *b, *c;
    printf ("Tamanho dos vetores: "); scanf ("%d", &m);
    a = (int *) malloc (m*sizeof(int));
    b = (int *) malloc (m*sizeof(int));
    c = (int *) malloc (m*sizeof(int));
    printf ("\nVetor a: ");
    for (i = 0; i < m; i++) scanf ("%d", &a[i]);
    printf ("\nVetor b: ");
    for (i = 0; i < m; i++) scanf ("%d", &b[i]);
    printf ("\nVetor c: ");
    for (i = 0; i < m; i++) c[i] = (a[i] > b[i])? a[i]: b[i];
    for (i = 0; i < m; i++) printf ("%5d", c[i]);
}
```

malloc retorna um ponteiro do tipo *void*; necessita conversão.

- A função *malloc* está na biblioteca *stdlib.h*
- Função *sizeof* (tp): retorna o n° de bytes necessários para armazenar uma variável do tipo **tp**:

```
sizeof (int) = 2
sizeof ( long) = 4
sizeof (char) = 1
sizeof (float) = 4
sizeof (double) = 8
```

```
struct st {
    int a; char c [10];};
typedef struct st st;
sizeof (st) = 12;
```

- Seja a declaração **float *a;** o comando

a = (float *) malloc (m*sizeof (float));

reserva um espaço de $m * \text{sizeof}(\text{float}) = m * 4$ bytes

faz **a** apontar para o 1º byte desse espaço

- O espaço alocado pertence a uma área especial do sistema, própria para essas reservas.
- Depois de totalmente usado o espaço, ele pode ser devolvido ao sistema pelo comando

free (a);

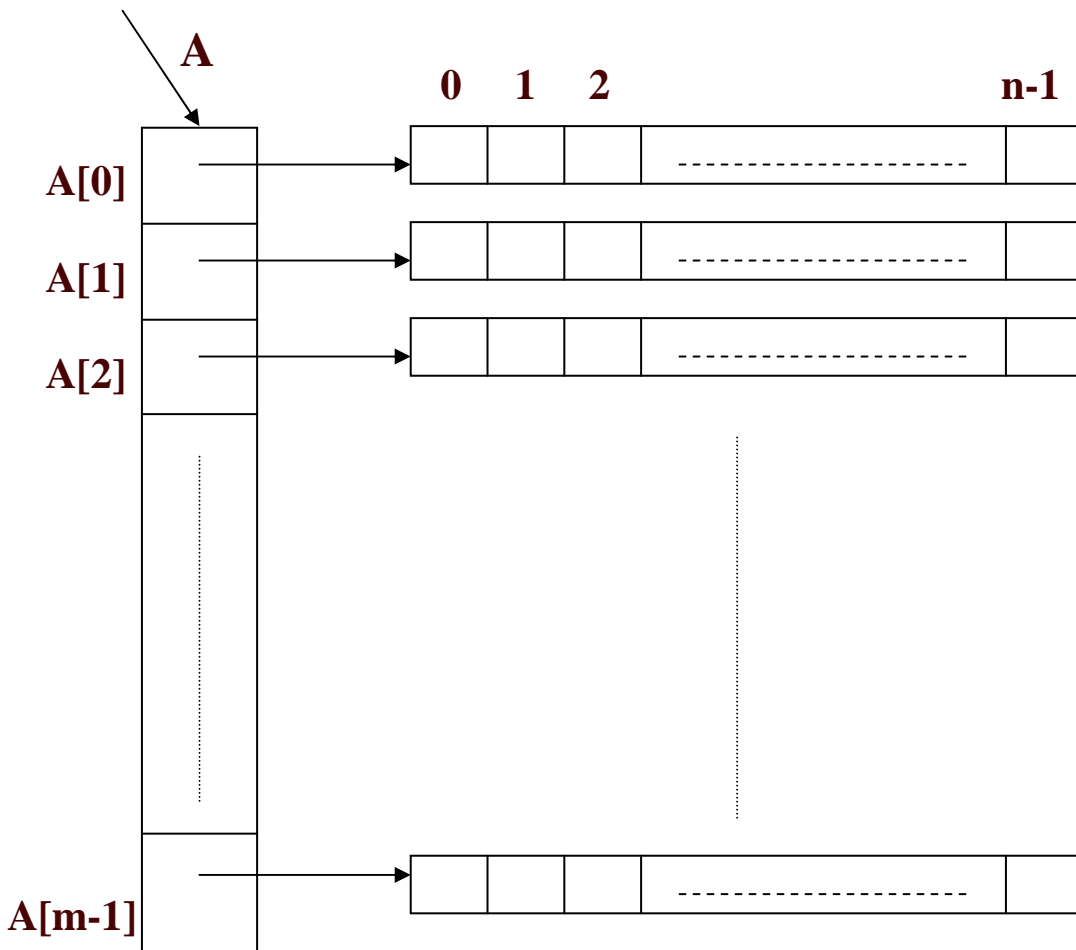
- Quando um programa faz muitas alocações dinâmicas, ele pode esgotar a capacidade dessa área especial.
- Se, durante a execução, algumas dessas alocações perderem sua utilidade, é bom que elas sejam devolvidas para o sistema; isso pode evitar tal esgotamento.

7.4 – Alocação Dinâmica de Matrizes

- A alocação dinâmica de matrizes pode feita por vetores de ponteiros para variáveis do tipo do elemento da matriz.
- **Exemplo 7.7:** seja o programa:

```
#include <stdio.h>
#include <stdlib.h>
typedef int *vetor;
typedef vetor *matriz;
void main () {
    int m, n, i, j; matriz A;
    printf ("Dimensoes da matriz: ");
    scanf ("%d%d", &m, &n);
    A = (vetor *) malloc (m * sizeof(vetor));
    for (i = 0; i < m; i++)
        A[i] = (int *) malloc (n * sizeof(int));
    printf ("\nElementos da matriz: \n");
    for (i = 0; i < m; i++) {
        printf ("\tLinha %d: ", i);
        for (j = 0; j < n; j++)
            scanf ("%d", &A[i][j]);
    }
    printf ("\nConfirmacao: \n");
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++)
            printf ("%5d", A[i][j]);
        printf ("\n");
    }
}
```

- Esquema da alocação dinâmica:



7.5 – Acesso a campos de Estruturas Apontadas

- Seja a estrutura:

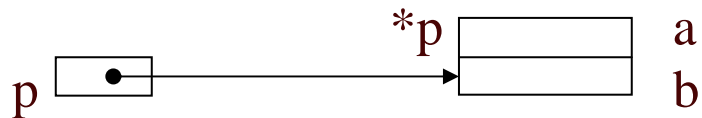
```
struct st { int a; float b; } ;  
typedef struct st st;
```

- Seja também um ponteiro para estruturas como essa:

```
st *p;
```

- Pode-se alocar dinamicamente espaço a ser apontado por **p**:

`p = (st *) malloc (sizeof (st));`



- Atribuições aos campos a e b da estrutura alocada:

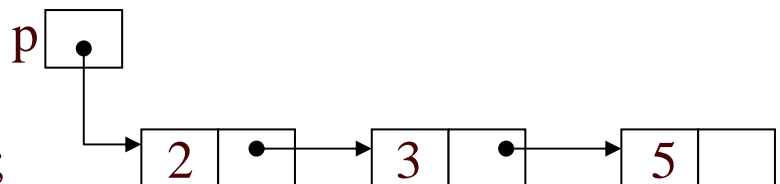
`(*p).a = 5; (*p).b = 17.3;`

- Há uma notação mais cômoda para essas atribuições

`p->a = 5; p->b = 17.3`

- Essa comodidade é útil em encadeamentos de estruturas (próximo capítulo);

```
struct st {
    int a ;
    struct st * prox;
}
typedef struct st st;
st *p;
```



`p-> prox -> prox -> a` é 5
ou
`(*(>(*p).prox).prox).a` é 5