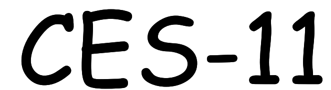
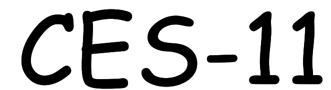
CES-11

Algoritmos e Estruturas de Dados

Carlos Alberto Alonso Sanches
Juliana de Melo Bezerra

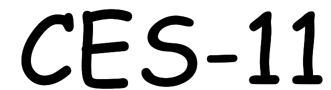


- Revisão
  - Tipos escalares primitivos
  - Tipos constituídos de uma linguagem
  - Ponteiros
  - Alocação estática *versus* dinâmica
  - Encadeamento de estruturas
  - Passagem de parâmetros
  - Recursividade



- Revisão
  - Tipos escalares primitivos
  - Tipos constituídos de uma linguagem
  - Ponteiros
  - Alocação estática *versus* dinâmica
  - Encadeamento de estruturas
  - Passagem de parâmetros
  - Recursividade

- Inteiro
- Real
- Lógico
- Caractere



- Revisão
  - Tipos escalares primitivos
  - Tipos constituídos de uma linguagem
  - Ponteiros
  - Alocação estática *versus* dinâmica
  - Encadeamento de estruturas
  - Passagem de parâmetros
  - Recursividade

Tipos constituídos de uma linguagem

- Vetores:

```
Tipo_primitivo  V[30], W[50], X[200];
```

ou

```
typedef   int   vetor[30];
vetor   V1, V2, V3;
```

- Matrizes:

```
Tipo_primitivo   M1[10][10][10],  M2[5][4];
```

ou

```
typedef   int  matriz[10][10];
matriz   M3, M4;
```

Vetores e matrizes são chamados de variáveis indexadas

Tipos constituídos de uma linguagem

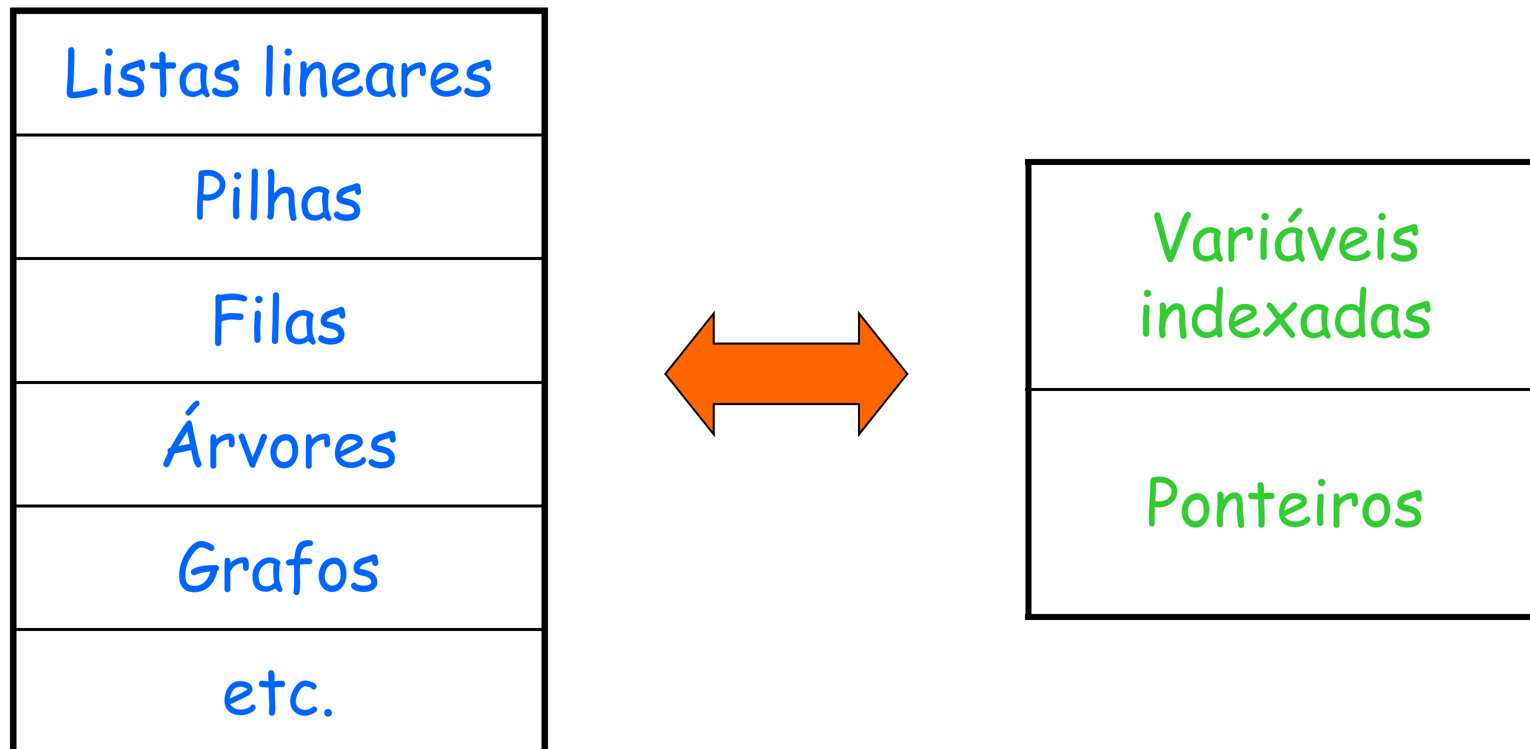
- Cadeias de caracteres:

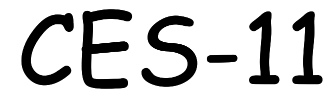
```
typedef   char  cadeia[15];
cadeia  nome, rua, aux;
```

- Estruturas simples:

```
typedef struct Funcionario Funcionario;
struct  Funcionario {
   char  nome[30], endereco[30], setor[15];
   char  sexo, estCivil;
   int  idade;
};

Funcionario  F1, F2, F3, empregados[200];
. . .
empregados[1] = F1;
F2.sexo = 'M';
strcpy (empregados[3].nome, "José da Silva");
```

Estruturas *versus* implementações



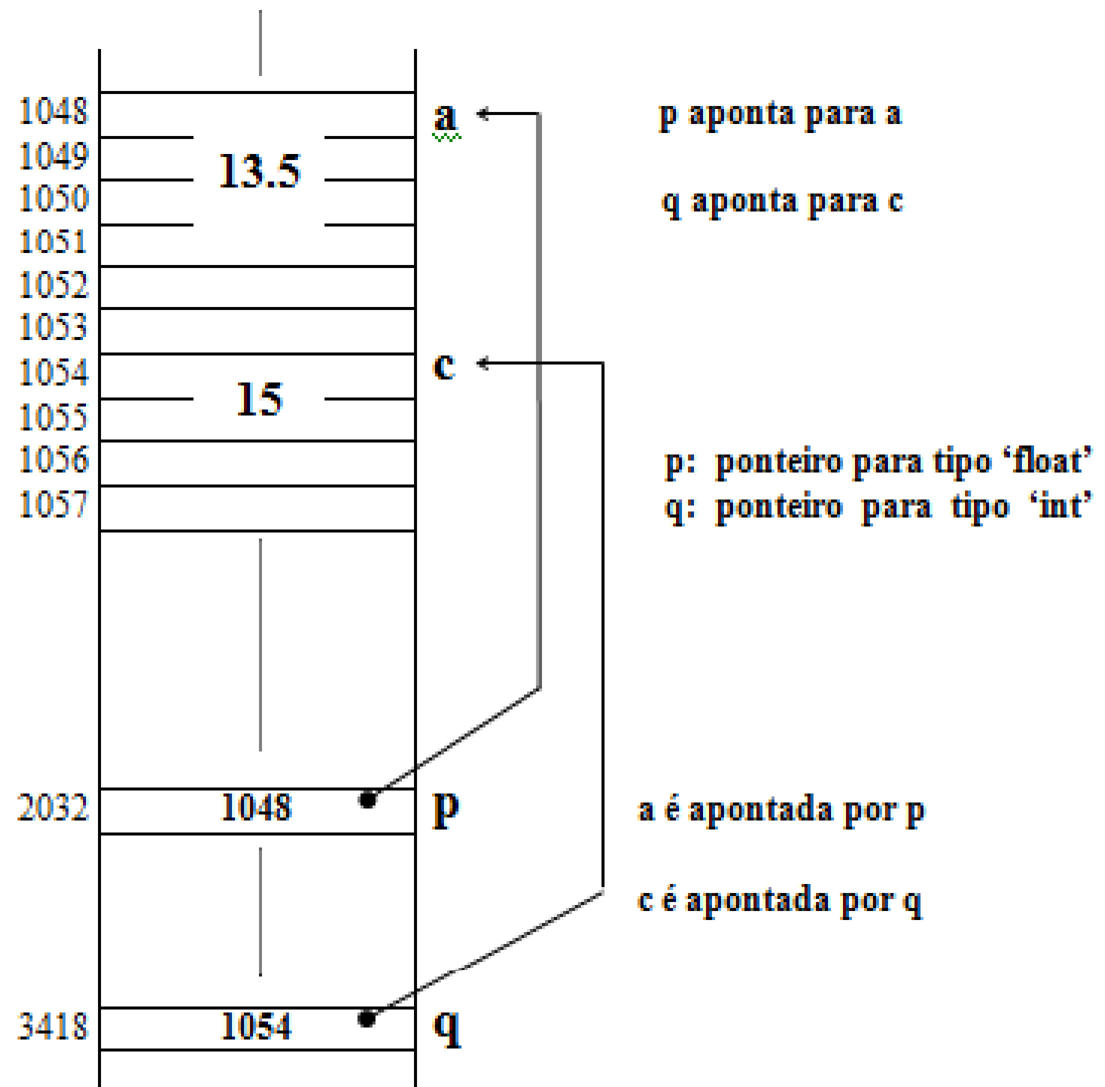


- Revisão
  - Tipos escalares primitivos
  - Tipos constituídos de uma linguagem
  - Ponteiros
  - Alocação estática *versus* dinâmica
  - Encadeamento de estruturas
  - Passagem de parâmetros
  - Recursividade

Ponteiros

- *Ponteiros* (ou *apontadores*) são variáveis que armazenam endereços de outras variáveis.
- No exemplo ao lado, `p` e `q` são ponteiros.
- Códigos:

```
float a; int c;
float *p; int *q;
p = &a; q = &c;
```



Ponteiros

- Principais utilidades de ponteiros:
  - Passagem de parâmetros por referência, em sub-programação
  - Alocação dinâmica de variáveis indexadas
  - Encadeamento de estruturas

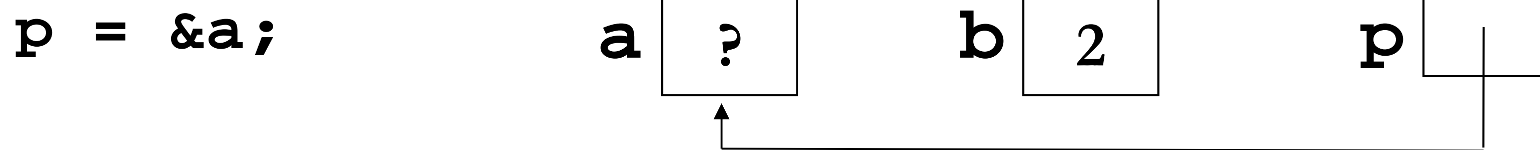
Ponteiros: notação

- Se `p` é um ponteiro, `*p` é o valor da variável apontada por `p`.
- Se `a` é uma variável, `&a` é o seu endereço.
- Exemplos:

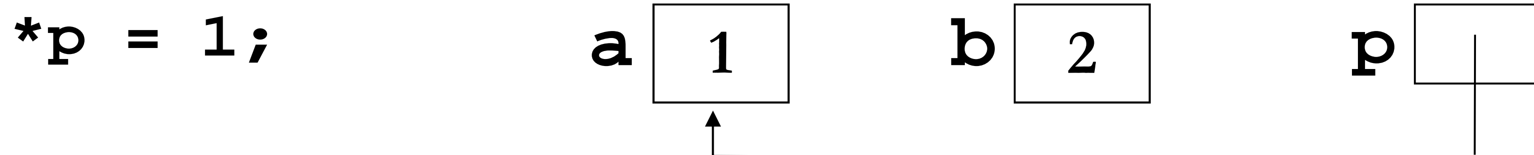
```
int a, b=2, *p;
```

a ? b 2 p ?

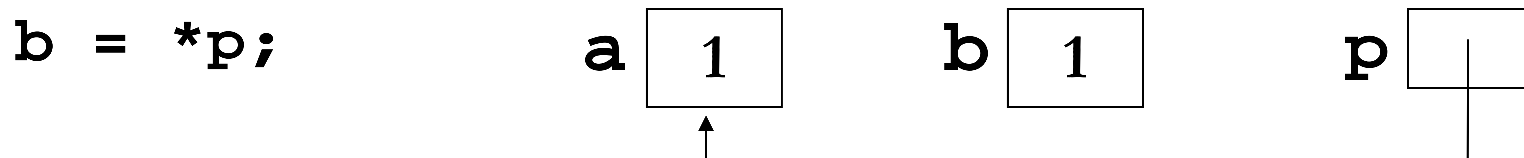
```
p = &a;
```




```
*p = 1;
```




```
b = *p;
```

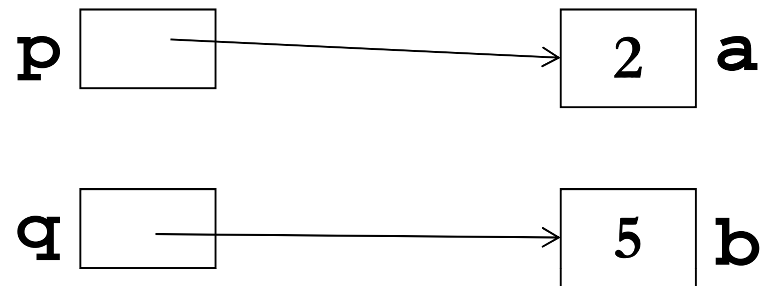


Ponteiros: exemplo

- Sejam as declarações abaixo:

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

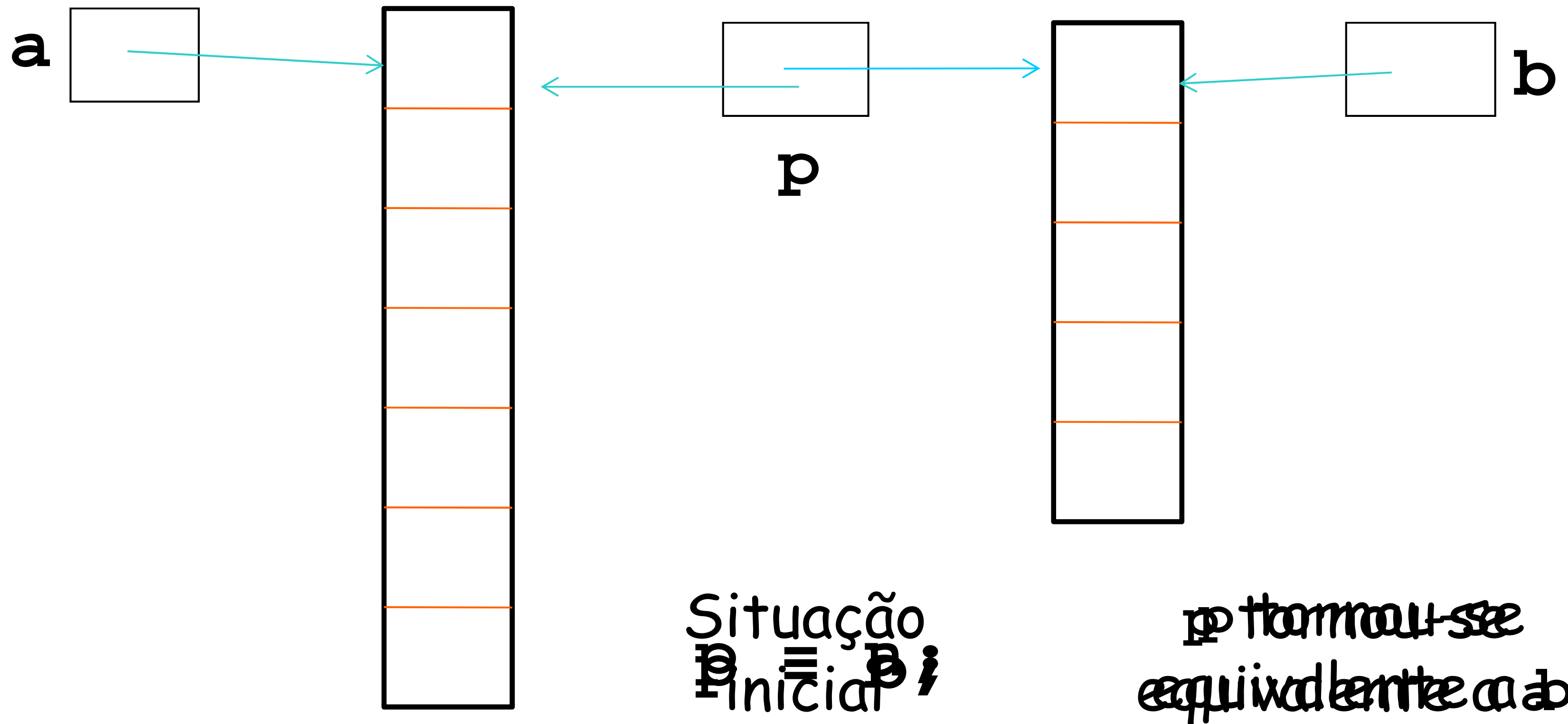
- Neste caso, a inicialização é de `p` e `q`, não de `*p` e `*q`:



Ponteiros e variáveis indexadas

- Sejam as declarações abaixo:

```
int a[7], *p, b[5];
```



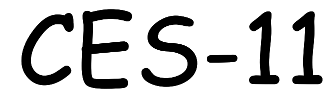
As atribuições `a = p` e `b = p` são proibidas!

Outras semelhanças

- Ponteiros podem ter índices, e variáveis indexadas admitem o operador unário *.
- Por exemplo, suponha as declarações abaixo: `int i, a[50], *p;`
  - `a[i]` é equivalente a `*(a+i)`
  - `*(p+i)` é equivalente a `p[i]`
- `a` contém o endereço de `a[0]`:
  - `p = a` equivale a `p = &a[0]`
  - `p = a+1` equivale a `p = &a[1]`

Qual é a diferença, então?

- Constante *versus* variável:
  - `a` é o endereço inicial de um vetor estático: seu valor não pode ser alterado
  - `p` é uma variável: seu conteúdo pode mudar
- Atribuições:
  - `p = &i` é permitido
  - `a = &i` não é permitido
- Endereços na memória:
  - `a[1]` tem sempre o mesmo endereço
  - `p[1]` pode variar de endereço



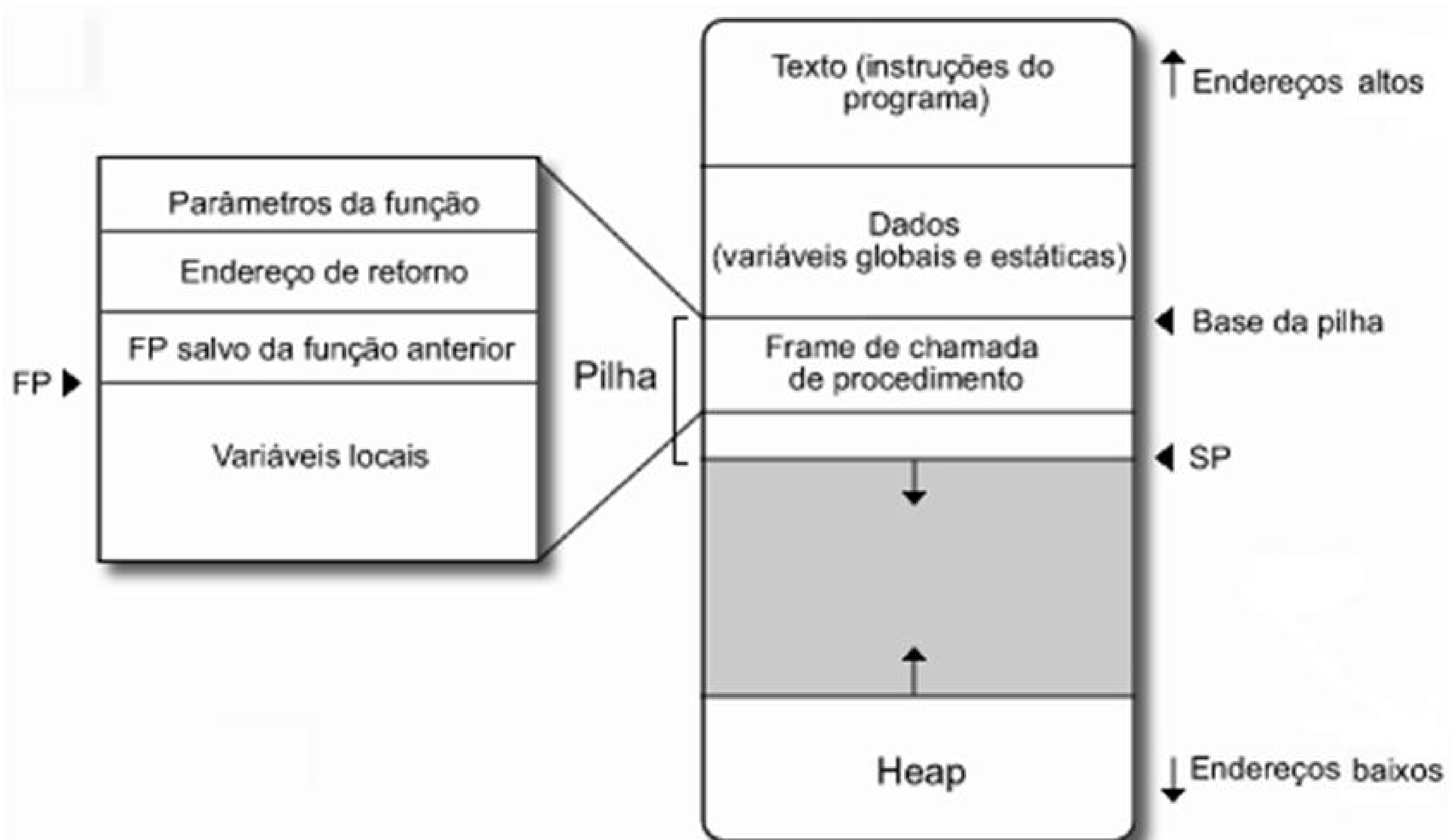
CES-11

- Revisão
  - Tipos escalares primitivos
  - Tipos constituídos de uma linguagem
  - Ponteiros
  - Alocação estática *versus* dinâmica
  - Encadeamento de estruturas
  - Passagem de parâmetros
  - Recursividade

Alocação estática *versus* dinâmica

- Variáveis estáticas: têm endereço determinado em *tempo de compilação*
  - São previstas antes da compilação do programa
  - Ocupam uma área de dados do programa, determinada na compilação
  - Existem durante toda a execução do programa
- Variáveis dinâmicas: têm endereço determinado em *tempo de execução*
  - São alocadas de uma área extra da memória, chamada *heap*, através de funções específicas (**`malloc`**, **`new`**, etc.)
  - Sua eventual existência depende do programa, e seu endereço precisa ser armazenado em outra variável
  - Exigem uma política de administração da memória

Pilha de execução



Alocação dinâmica de memória

- Muitas vezes, é conveniente alocar espaço para uma variável indexada apenas em tempo de execução.
- Nesse caso, essa variável deve ser inicialmente alocada como ponteiro.
- Durante a execução do programa, o espaço de memória necessário para essa variável pode ser alocado através da função `malloc`.

Exemplo

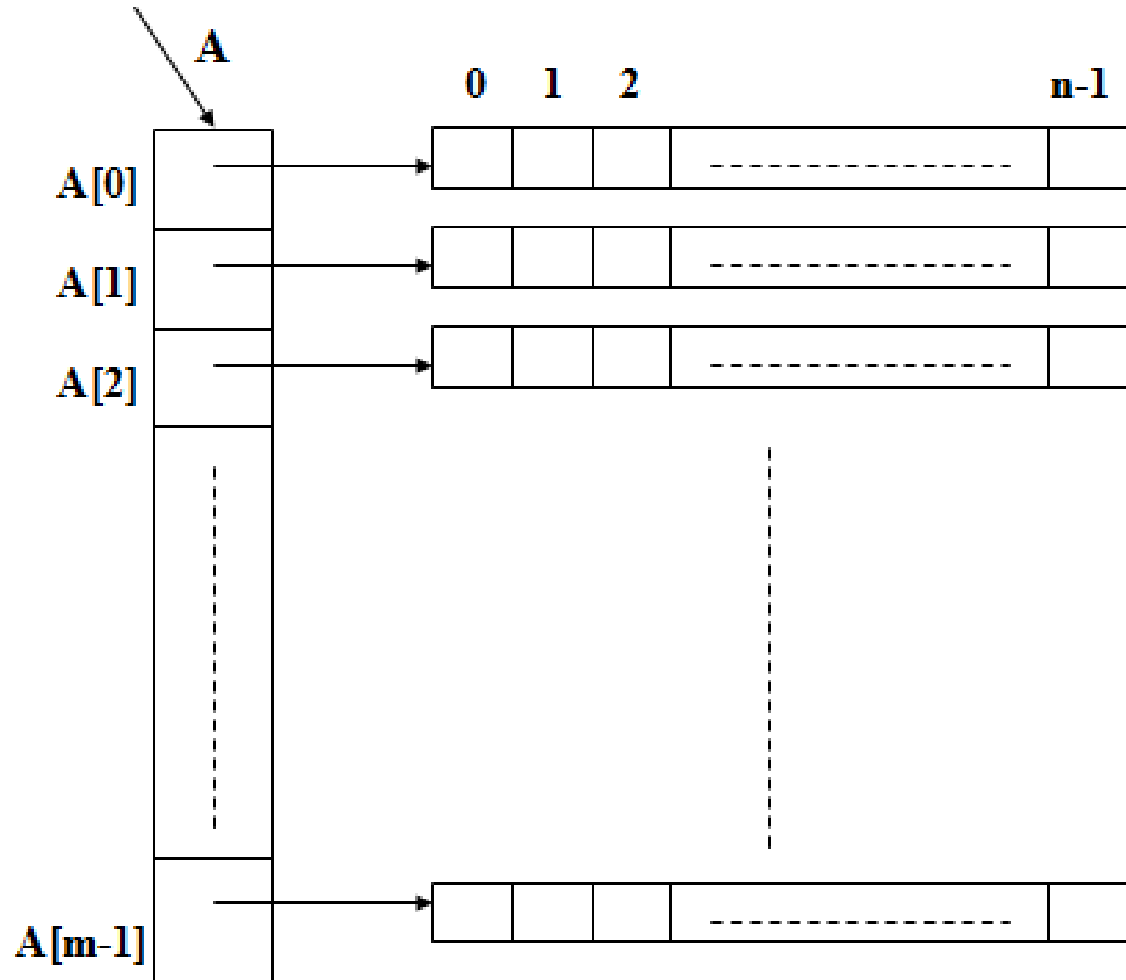
```
typedef int *vetor;
void main () {
   int m, i; vetor A, B, C;
   printf("Tamanho dos vetores: ");
   scanf("%d",&m);
   A = (int *) malloc (m*sizeof(int));
   B = (int *) malloc (m*sizeof(int));
   C = (int *) malloc (m*sizeof(int));
   printf("Vetor A: ");
   for (i = 0; i < m; i++) scanf("%d",&A[i]);
   printf("Vetor B: ");
   for (i = 0; i < m; i++) scanf("%d",&B[i]);
   printf("Vetor C: ");
   for (i = 0; i < m; i++)
       C[i] = (A[i] > B[i])? A[i] : B[i];
   for (i = 0; i < m; i++) printf("%d",C[i]);
}
```

m: dimensão do vetor (definida em tempo de execução)

C[i] será o maior entre A[i] e B[i]

Alocação dinâmica de matrizes

- Uma matriz também pode ser alocada em tempo de execução, de modo análogo aos vetores.
- Exemplo: matriz m x n.



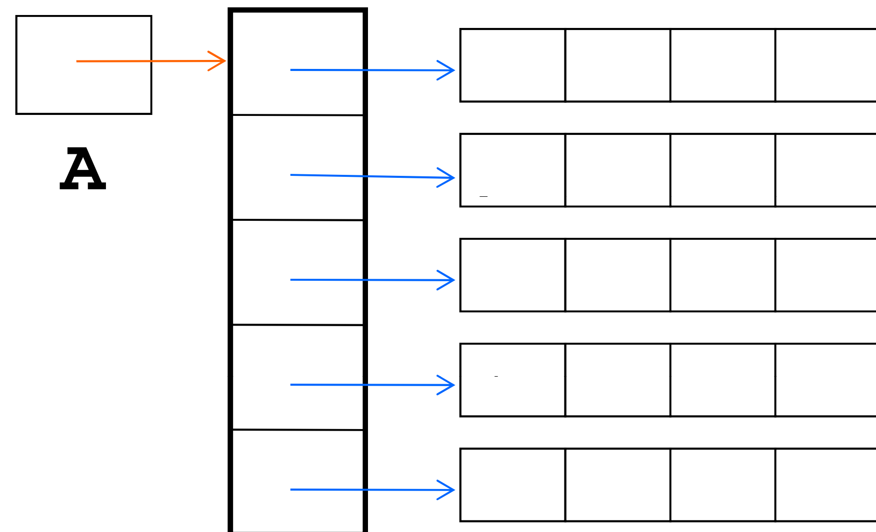
Gasta-se mais espaço: um ponteiro para cada linha

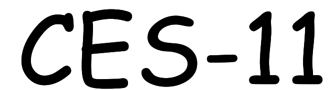
Exemplo

```
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("Elementos da matriz:");
   for (i = 0; i < m; i++) {
        printf("Linha %d ", i);
        for (j = 0; j < n; j++) scanf("%d",&A[i][j]);
   }
}
```

Dimensões da matriz:

m 5 n 4



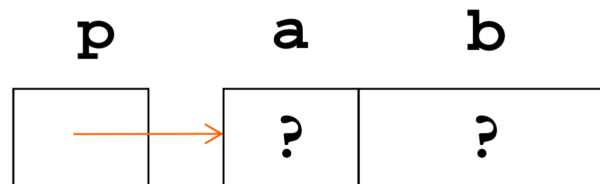


- Revisão
  - Tipos escalares primitivos
  - Tipos constituídos de uma linguagem
  - Ponteiros
  - Alocação estática *versus* dinâmica
  - Encadeamento de estruturas
  - Passagem de parâmetros
  - Recursividade

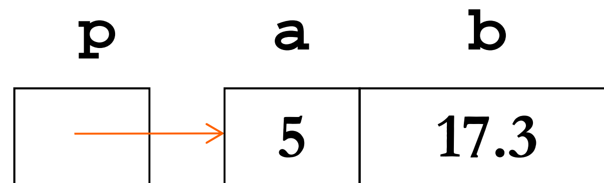
Encadeamento de estruturas

- Considere o código abaixo:

```
struct st {int a; float b};
st *p;
p = (st *) malloc (sizeof(st));
```




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

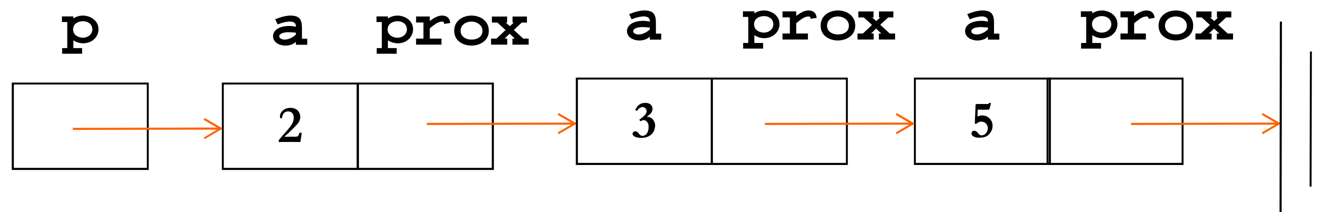


- Código equivalente às atribuições acima:

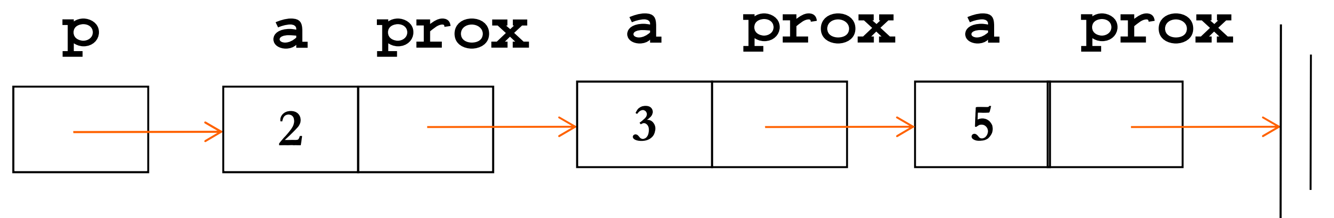
```
p->a = 5; p->b = 17.3;
```

Outro exemplo

```
struct noh {int a; noh *prox};
noh *p;
p = (noh *) malloc (sizeof(noh));
p->a = 2;

p->prox = (noh *) malloc (sizeof(noh));
p->prox->a = 3;

p->prox->prox = (noh *) malloc (sizeof(noh));
p->prox->prox->a = 5;

p->prox->prox->prox = NULL;
```



Continuando



- Escrita do campo a de todos os nós:

```
noh *q;
for (q=p; q!=NULL; q=q->prox)
                printf("%d",q->a);
```

- Acesso ao campo a do último nó:

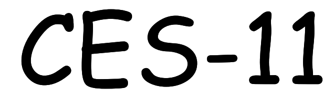
```
p->prox->prox->a
```
← mais simples

ou

```
(*(*(*p).prox).prox).a
```

Encadeamento de estruturas

- Baseia-se na utilização de variáveis ponteiros
- Proporciona muitas alternativas para estruturas de dados
- É usado em listas lineares, árvores e grafos



- Revisão
  - Tipos escalares primitivos
  - Tipos constituídos de uma linguagem
  - Ponteiros
  - Alocação estática *versus* dinâmica
  - Encadeamento de estruturas
  - Passagem de parâmetros
  - Recursividade

Passagem de parâmetros

- Declaração de funções:

Tipo *Nome_de_função* (*Lista_de_parâmetros*) {
Corpo_de_função
}

  - Funções que não retornam valores são do tipo **void**
  - A lista de parâmetros pode ser vazia ou não
  - Parâmetros sempre são alocados dinamicamente, e recebem os valores que lhe são passados na chamada

Duas formas de passagem: por valor ou por referência

Passagem de parâmetros

- Passagem por **valor**

```
void ff (int a) {
        a += 1;
        printf ("Durante ff: a = %d \n", a);
}

void main ( ) {
        int a = 5;
        printf ("Antes de ff: a = %d \n", a);
        ff (a);
        printf ("Depois de ff: a = %d \n", a);
}
```

a 6

a 5

Saída:

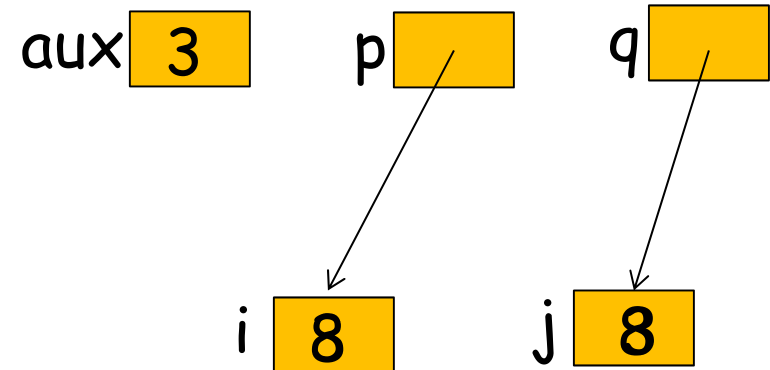
```
Antes de ff: a = 5
Durante ff: a = 6
Depois de ff: a = 5
```

outra variável!

Passagem de parâmetros

- Passagem por **referência**

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

void main ( ) {
    int i = 3, j = 8;
    printf ("Antes: i = %d, j = %d \n", i, j);
    trocar (&i, &j);
    printf ("Depois: i = %d", i, j);
}
```



Saída:

```
Antes: i = 3, j = 8
Depois: i = 8, j = 3
```

<u>Outra vantagem</u>: economiza memória ao se trabalhar com grandes estruturas

Passagem de parâmetros

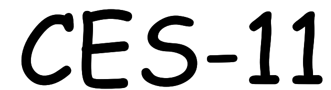
- Passagem por **referência**
  - *Variável indexada como parâmetro*

```
#include <stdio.h>
void alterar (int B[]) {
   B[1] = 5;
   B[3] = 5;
}
void main ( ) {
   int i, j, A[10] = {0};
   //imprimir vetor A
   alterar(A);
   //imprimir vetor A
   alterar(&A[4]);
   //imprimir vetor A
 }
```

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

0	1	2	3	4	5	6	7	8	9
0	5	0	5	0	0	0	0	0	0

0	1	2	3	4	5	6	7	8	9
0	5	0	5	0	5	0	5	0	0



- Revisão
  - Tipos escalares primitivos
  - Tipos constituídos de uma linguagem
  - Ponteiros
  - Alocação estática *versus* dinâmica
  - Encadeamento de estruturas
  - Passagem de parâmetros
  - Recursividade

Recursividade

- Uma função é <u>recursiva</u> se fizer alguma chamada a si mesma.
- Ex1: soma dos n primeiros números naturais

```
int soma (int n) {
    int i, resultado = 0;
    for (i=1; i<=n; i++)
        resultado = resultado + i;
    return resultado;
}
```

```
int somaRecursiva (int n) {
    if (n==1) return 1;
    return n + somaRecursiva(n-1);
}
```

Cuidado com *loop* infinito

Recursividade

- Ex2: cálculo de potência

$$A^n = \text{Power}(A, n) = \begin{cases} 1, & \text{se } n = 0 \\ A, & \text{se } n = 1 \\ A * \text{Power}(A, n-1), & \text{se } n > 1 \end{cases}$$

- Ex3: cálculo de fatorial de números positivos

$$\text{Fat}(n) = \begin{cases} -1 & \text{se } n < 0 \\ 1 & \text{se } n = 0 \text{ ou } n = 1 \\ n * \text{Fat}(n - 1) & \text{se } n > 1 \end{cases}$$

Recursividade

- Ex4: máximo divisor comum (algoritmo de Euclides)

$$MDC(m, n) = \begin{cases} m & \text{se } n = 0 \\ MDC(n, m \% n), & \text{se } n > 0 \end{cases}$$

42 = 2 * 3 * 7
30 = 2 * 3 * 5

MDC(42,30) = 6

m = n * q + r

MDC(42,30)	42 = 30 * 1 + 12
MDC(30,12)	30 = 12 * 2 + 6
MDC(12,6)	12 = 6 * 2 + 0
MDC(6,0)	Retorna 6

- Será que funciona se calcularmos MDC(30,42)?

Recursividade

- Ex4: máximo divisor comum (algoritmo de Euclides)

$$MDC(m, n) = \begin{cases} m & \text{se } n = 0 \\ MDC(n, m \% n), & \text{se } n > 0 \end{cases}$$

42 = 2 * 3 * 7

30 = 2 * 3 * 5

MDC(30,42) = 6

m = n * q + r

MDC(30,42)	30 = 42 * 0 + 30
MDC(42,30)	42 = 30 * 1 + 12
MDC(30,12)	30 = 12 * 2 + 6
MDC(12,6)	12 = 6 * 2 + 0
MDC(6,0)	Retorna 6

Recursividade

- Ex5: busca binária em vetor ordenado

Procura (Elemento, Vetor, inf, sup) =

- -1, se Elemento < Vetor [inf] ou
  se Elemento > Vetor [sup]
- médio, se Elemento = Vetor [medio]
- Procura (Elemento, Vetor, inf, medio - 1),
  se Elemento < Vetor [medio]
- Procura (Elemento, Vetor, medio + 1, sup),
  se Elemento > Vetor [medio]

medio = (inf + sup) /2

Recursividade

- Exemplo: Procura(28, vet, 0, 9)