

ALGORITMOS E ESTRUTURAS DE DADOS CES-11

Prof. Paulo André Castro
pauloac@ita.br
Sala 110 - Prédio da Computação
www.comp.ita.br/~pauloac
IECE - ITA

CES-11

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

Tipos escalares primitivos

○ Tipo **Inteiro**:

- Domínio: números inteiros entre $-\infty$ e $+\infty$
- Operações: + - * / % ++ -- = ≠ < ≤ > ≥
- Exemplos: $7 / 3 = 2$ $7 \% 3 = 1$

○ Tipo **Real**:

- Domínio: números reais entre $-\infty$ e $+\infty$
- Operações: + - * / = ≠ < ≤ > ≥

Tipos escalares primitivos

○ Tipo **Lógico**:

- Domínio: **Verdadeiro** e **Falso**
- Operações: =, ≠, **and**, **or**, **not**, **nand**, **nor** e **xor**
- Os resultados das comparações são valores lógicos

○ Tipo **Caractere**:

- Domínio:
 - Dígitos decimais: '0', '1', ..., '9'
 - Letras: 'A', 'B', ..., 'Z', 'a', 'b', ..., 'z'
 - Sinais especiais: '.', ',', ';', '+', '-', '*', '/', '(', ')', '[', ']', '!', '...', ...
- Operações: = ≠ (< ≤ > ≥ + - * / %)

CAPACIDADE DE REPRESENTAÇÃO – LINGUAGEM C

Inteiros:

Tipo	N.o de bytes	Intervalo de valores
int	2	-32768 a +32767
short	2	-32768 a +32767
long	4	-2147483648 a +2147483647
unsigned	2	0 a 65535

Ponto Flutuante(Reais):

tipo	precisão	intervalo
float	6 dígitos significativos	-38 a +38
double	15 dígitos significativos	-308 a +308

• Números reais não são representados com exatidão;

○ O número 3426175.8390176294015 (20 dígitos significativos) é representado como:

float: 0.342617 *10⁷
double: 0.342617583901762 *10⁷

CES-11

- 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:

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

```
Funcionario F1, F2, F3, empregados[200];
```

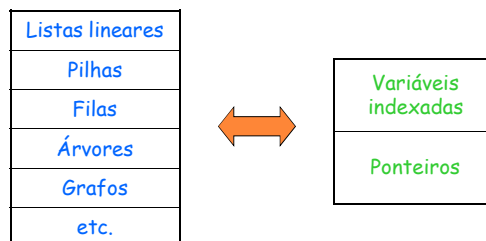
```
...
empregados[1] = F1;
F2.sexo = 'M';
strcpy (empregados[3].nome, "José da Silva");
```

Tipos constituídos de uma linguagem

○ Outros:

- Estruturas de campos alternativos (UNION)
- Ex.: union int_or_float { int i; float f; }
- Tipos enumerativos (ENUM)
- Ex.: enum diasemana { dom, seg, ter, qua, qui, sex, sab};

Estruturas *versus* implementações



○ Revisão

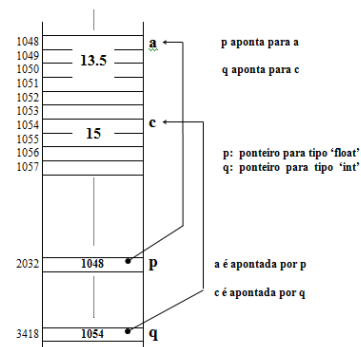
CES-11

- 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.

■ Declarações:
float a; int c;
float *p; int *q;
p = &a; q = &c;

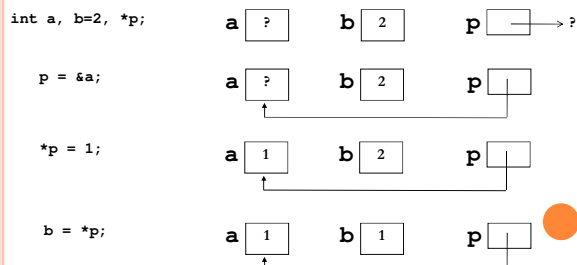


PONTEIROS

- 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:

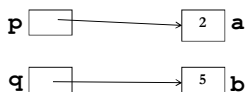


PONTEIROS: EXEMPLO

- Sejam as declarações abaixo:

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

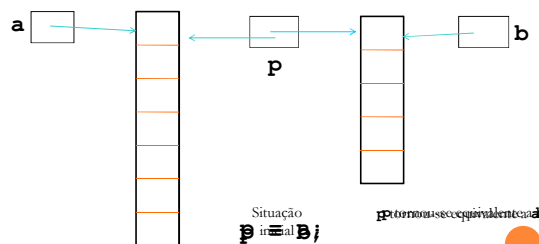
- A inicialização é de p e q , não de $*p$ e $*q$.



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

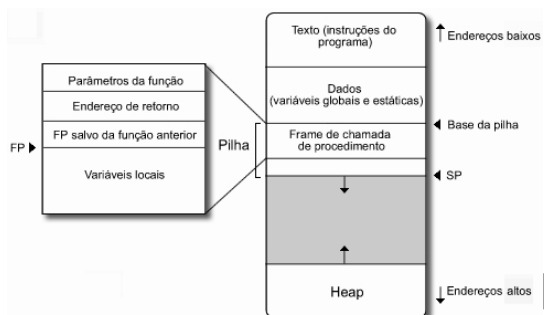
○ **Variáveis estáticas:** têm endereço determinado em *tempo de compilação*

- 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
- Exige uma política de administração da memória

PILHA DE EXECUÇÃO



ALOCAÇÃO DINÂMICA DE MEMÓRIA

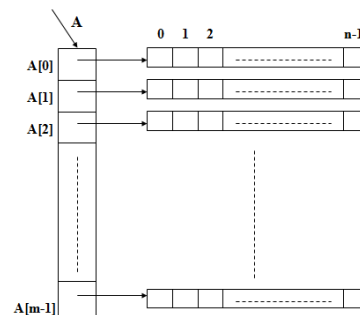
- Muitas vezes é interessante alocar memória apenas em tempo de execução, neste caso deve-se utilizar ponteiros
- Durante a execução do programa, o espaço de memória necessário para essa variável pode ser alocado através da função `malloc`.

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]);
}
```

ALOCAÇÃO DINÂMICA DE MATRIZES

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



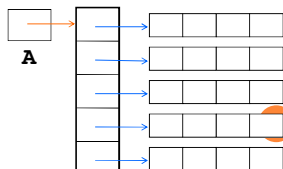
Gasta-se mais espaço: um ponteiro para cada linha

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:



CES-11

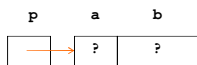
- Revisão
 - Tipos escalares primitivos
 - Tipos constituídos de uma linguagem
 - Ponteiros
 - Alocação estática *versus* dinâmica
 - Encadeamento de estruturas
 - Passagem de parâmetros
 - Recursividade

ENCADEAMENTO DE ESTRUTURAS

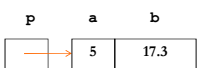
- Considere o código abaixo:

```

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;
    
```

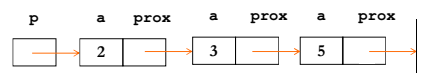
```

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

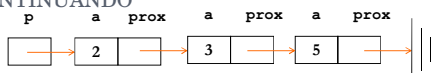
```

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

CES-11

- 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);
}
```

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, j = %d", i, j);
}
```

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

Outra vantagem: economia de 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
B[]	0	0	0	0	0	0	0	0	0	0
A[]	0	5	0	5	0	0	0	0	0	0
A[4]	0	5	0	5	0	5	0	5	0	0

CES-11

- 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 **recursiva** 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;
}
    
```

Mais elegante!

```

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

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

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

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

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

- Ex5: busca binária em vetor ordenado
- Procura(28, vet, 0, 9)

	inf		medio	inf		sup	medio				sup
	0	1	2	3	4	5	6	7	8	9	
vet:	5	12	28	37	46	51	66	74	82	93	

Achou!

Recursividade

○ Ex6: reconhecimento de cadeias de caracteres

- Uma cadeia contendo apenas uma letra ou dígito é válida.
- Se α é uma cadeia válida, então (α) também será.

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

typedef char cadeia[30];
cadeia cad;
int i;
bool erro;

void testarCadeia (void);
```

Contém printf
Contém getche () e gets ()
Contém operações para string
Variáveis globais
Protótipo de função

Recursividade

○ Ex6: reconhecimento de cadeia de caracteres

```
void main() {
    char c;
    printf ("Testar cadeia? (s/n): ");
    do c = getche ();
    while (c!='s' && c!='n');
    while (c == 's') {
        clrscr (); printf ("Digite a cadeia: ");
        fflush (stdin); gets (cad);
        i = 0; erro=false;
        testarCadeia ();
        if (cad[i] != '\0') erro = true;
        if (erro) printf ("cadeia reprovada!");
        else printf ("cadeia valida!");
        printf ("\n\nTestar nova cadeia? (s/n): ");
        do c = getche ();
        while (c!='s' && c!='n'); } }
```

Recursividade

○ Ex6: reconhecimento de cadeia de caracteres

```
void testarCadeia () {
    if (isalpha(cad[i]) || isdigit(cad[i]))
        i++;
    else if (cad[i] == '(') {
        i++;
        testarCadeia ();
        if (erro==false && cad[i] == ')')
            i++;
        else erro = true;
    }
    else erro = true;
}
```

Recursividade

○ Ex6: reconhecimento de cadeia de caracteres

- O código funciona, mas não é elegante.
- Pontos fracos:
 - Não verifica se a cadeia excede 30 caracteres
 - `cad`, `i` e `erro` são variáveis globais
 - O ideal seria: `bool testarCadeia(cadeia, indice)`
 - Para que o código abaixo está na função `main`?

```
if (cad[i] != '\0') erro = true;
```

 - Gera erro em casos com 2 letras/números seguidos. Ex: aa, (bb)
 - Isso deveria ser escopo da função `testarCadeia`.

Recursividade

○ Ex7: reconhecimento de expressões aritméticas

- Uma expressão com apenas uma letra ou dígito é válida.
- Se α e β são expressões válidas, então $(\alpha+\beta)$, $(\alpha-\beta)$, $(\alpha*\beta)$ e (α/β) também serão.

○ Ex8: cadeias com n zeros iniciais ($n \geq 0$) seguidos de $2n$ um's

- Uma cadeia vazia é válida.
- Se α é uma cadeia válida, então $0\alpha11$ também é.