

**INSTITUTO FEDERAL**

Bahia

Campus Santo Antônio de Jesus

<http://www.portal.ifba.edu.br/santoantonio>

# LINGUAGEM DE PROGRAMAÇÃO

**Prof. George Pacheco Pinto**

# AGENDA

- ❑ Linguagem C
  - ❑ Ponteiros
    - ❑ Conceito
    - ❑ Declaração
    - ❑ Manipulação

# VARIÁVEIS

- ❑ Tudo aquilo que é sujeito a variações, incerto, instável ou inconstante.
- ❑ Podem conter valores diferentes a cada instante de tempo. Seu valor pode ser alterado ao longo do tempo da execução do programa;
- ❑ Só pode assumir um único valor a cada instante.

# VARIÁVEIS

- ❑ Variável = nome + tipo + valor
- ❑ A criação de uma variável leva a atribuição de um bloco específico da memória do computador para guardar seu valor;
- ❑ O tamanho do bloco depende dos valores permitidos para a variável. Ex: `int` ocupa 4 bytes
- ❑ `sizeof(int)` – descobre o tamanho do tipo inteiro no sistema.

# VARIÁVEIS

```
int k;
```

# VARIÁVEIS

```
int k;
```

4 bytes são separados e uma tabela de símbolos é montada. Nela coloca-se o símbolo K e o endereço na memória dos 4 bytes

# VARIÁVEIS

4 bytes são separados e uma tabela de símbolos é montada. Nela coloca-se o símbolo K e o endereço na memória dos 4 bytes

```
int k;
```

`k = 2;` *Duas informações associadas*

lvalue      rvalue

# VARIÁVEIS

4 bytes são separados e uma tabela de símbolos é montada. Nela coloca-se o símbolo K e o endereço na memória dos 4 bytes

```
int k;
```

`k = 2;` *Duas informações associadas*

lvalue      rvalue

- ❑ O lvalue (endereço) aparece do lado esquerdo do operador '=' e o rvalue (valor) aparece do lado direito. Assim `2 = k` é ilegal.



# VARIÁVEIS

Exemplo

```
int j, k;  
k=2;
```

```
j=7;
```

```
k=j;
```

# VARIÁVEIS

## Exemplo

```
int j, k;  
k=2;
```

```
j=7;
```

Aqui o compilador interpreta j como sendo o endereço da variável (lvalue)

```
k=j;
```

Aqui j é interpretado como seu rvalue. Copia valor de j para k.

# ENTÃO

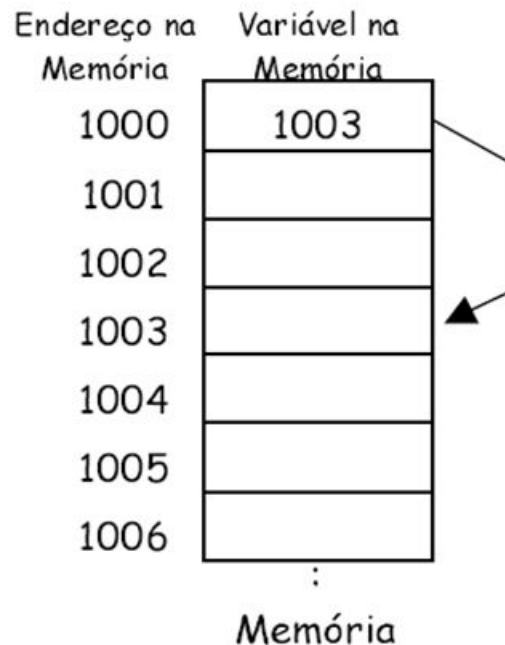
- ❑ Se por algum acaso quisermos armazenar em uma variável **endereço**, o que fazer?

Ou seja,

- ❑ Na expressão  $k = j$ , ao invés do valor 7,  $k$  recebesse o endereço de  $j$ .

# PONTEIROS

- ❑ Um ponteiro é uma variável cujo conteúdo é um endereço de memória;
- ❑ Esse endereço normalmente é a posição de uma outra variável na memória;
- ❑ Se uma variável contém o endereço de uma outra, então a primeira variável é dita apontar para a segunda.



SCHILD (1997)

# PONTEIROS

- ❑ Variáveis comuns referenciam um valor específico;
- ❑ Ponteiros contém o endereço de uma variável com um valor específico.

cont

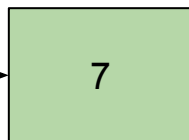


cont faz uma referência direta a uma variável cujo valor é 7.

contPtr



cont



contPtr faz uma referência indireta a uma variável cujo valor é 7.

# PONTEIROS

- ❑ Permitem simular chamadas de funções por referência;
- ❑ Manipulação de vetores e strings;
- ❑ Permitem criar e manipular estruturas dinâmicas de dados;
  - ❑ Filas, Listas encadeadas, Pilhas e árvores

# DECLARAÇÃO DE PONTEIROS

- ❑ Como toda variável, um ponteiro deve ser declarado.
- ❑ A declaração de uma variável do tipo ponteiro consiste do tipo base (aquele para o qual o ponteiro vai apontar), um \* e o nome da variável.
- ❑ A forma geral é:  
tipo \*nome;  
ou  
tipo\* nome;

# DECLARAÇÃO DE PONTEIROS

Exemplos:

```
int  *contador;    //ponteiro para um inteiro
```

```
char  *meuString;    //ponteiro para caracteres
```

```
float  *raizQuadrada; //ponteiro para real.
```



# OPERADORES DE PONTEIROS

- ❏ & - devolve o endereço na memória de seu operando;

- ❏ Ex:

```
int *m, cont;
```

```
m = &cont;    //coloca em m o endereço da memória  
               reservado para variável cont.
```

- ❏ & = retorna “o endereço de”

- ❏ “m recebe o endereço de cont”

# OPERADORES DE PONTEIROS

- ❑ \* (indireção) – devolve o valor da variável localizada no endereço que o segue;
- ❑ Ex:  
int q;  
q = \*m;    //coloca o valor de cont em q
- ❑ \* = “no endereço”
  - ❑ “q recebe o valor que está no endereço m”

# EXEMPLOS

```
int a = 5, *p = &a;  
printf ("%d\n", a);  
printf ("%d\n", *p);  
*p = 7;  
printf ("%d", a);
```

```
int x, *p;  
x = 2;  
p = &x;  
printf ("%p\n", p);
```

# EXEMPLOS

```
int main (){
    int *p;
    int valor1, valor2;

    valor1 = 5; //inicializa valor1 com 5
    p = &valor1; //p recebe o endereco de valor1
    valor2 = *p; //valor2 recebe o valor apontado por p, nesse caso 5

    printf ("P aponta para %p\n", p);
    printf ("P contem o valor %d\n", *p);
    printf ("O endereço de P %p\n", &p);
}
```

# EXEMPLOS

```
int main (){  
    int *p;  
    int valor1, valor2;  
  
    valor1 = 5; //inicializa valor1 com 5  
    p = &valor1; //p recebe o endereco de valor1  
    valor2 = *p; //valor2 recebe o valor apontado por p, nesse caso 5  
  
    printf ("P aponta para %p\n", p);  
    printf ("P contem o valor %d\n", *p);  
    printf ("O endereço de P %p\n", &p);  
}
```



O operador de endereço &, quando usado como operador sobre um ponteiro, devolve o endereço ocupado por este ponteiro, não o endereço apontado por ele!!!

# ATENÇÃO



- ❑ Variáveis ponteiros sempre devem apontar para o tipo de dado correto.

```
int main (){  
    float x, y;  
    int *p;  
  
    p = &x; //faz com que p aponte para um float  
    y = *p; //não funciona como esperado  
}
```

# ARITMÉTICA DE PONTEIROS

- ❑ "C" permite que se faça uma série de operações utilizando ponteiros, inclusive operações aritméticas, como soma e subtração, além de comparações entre ponteiros.
- ❑ Isto é muito útil, pode porém, ser também muito perigoso, pois permite ao programador uma liberdade que em nenhuma outra linguagem de programação é possível.

# ATRIBUIÇÃO

- ❑ Atribuição direta entre ponteiros passa o endereço de memória apontado por um para o outro

```
int *p1, *p2, x;  
x = 4;  
p1 = &x;  /* p1 passa a apontar para x */  
p2 = p1;  /* p2 recebeu o valor de p1, que é o endereço de x, ou seja: p2  
           também aponta para x.  */  
  
printf ("%p", p2 ); /* imprime o endereço de x */  
printf ("%d", *p2 ); /* imprime o valor apontado por p2, ou seja: o  
                     valor de x. */
```



# ARITMÉTICA DE PONTEIROS

- ❑ Duas operações aritméticas são válidas com ponteiros: adição e subtração. Estas são muito úteis com vetores.

```
int *p1, *p2, *p3, *p4, x=0;  
p1 = &x;  
p2 = ++p1;  
p3 = p2 + 4;  
p4 = p3 - 5;
```

p4 acaba tendo o mesmo valor que p1 no começo. Note que p1 foi incrementado e agora tem o valor (&x + 1).

Observe que aqui as expressões \*p1, \*p2 e \*p3 vão resultar em um erro (pega lixo), já que esses ponteiros estarão apontando para áreas de memória que não estão associadas com nenhuma variável. O único endereço de memória acessível é o de x.

# ARITMÉTICA DE PONTEIROS

- ❑ Para o cálculo do incremento ou decremento é usado sempre o TAMANHO DO TIPO BASE DO PONTEIRO.
- ❑ Isto significa que se `p1` aponta para o endereço 2000, `p1 + 2` não necessariamente vai ser igual a 2002. Se o tipo base é um inteiro (`int *p1`), que em Unix sempre possui tamanho 4 bytes, então `p1 + 2` é igual a 2008.
  - ❑ Ou seja: o valor de `p1` adicionado de duas vezes o tamanho do tipo base.

# ARITMÉTICA DE PONTEIROS

- ❑ No exemplo anterior, se o endereço de  $x$  é 1000:
  - ❑  $p1$  recebe o valor 1000, endereço de memória de  $x$ .
  - ❑  $p2$  recebe o valor 1004 e  $p1$  tem seu valor atualizado para 1004.
  - ❑  $p3$  recebe o valor  $1004 + 4 * 4 = 1020$ .
  - ❑  $p4$  recebe o valor  $1020 - 5 * 4 = 1000$ .
- ❑ Se as variáveis acima fossem do tipo char (1 byte de tipo base), os endereços seriam, respectivamente: 1000, 1001, 1001, 1005 e 1000.

# PONTEIROS E VETORES

- ❑ Ponteiros e Vetores possuem uma relação muito estreita em "C".

- ❑ Ex:

```
int x[80], y;  
int *p1;
```

```
p1 = &x[0];    // p1 é inicializado com a primeira  
                posição do vetor x
```

# PONTEIROS E VETORES

```
y = *p1           // y recebe o conteúdo de x[0]

p1 + 1            // aponta para o elemento seguinte do vetor

*p1 + 1           // soma 1 ao elemento apontado por p1

*(p1 + 1)         // refere-se ao conteúdo da próxima posição
                  // apontada por p1
```

# PONTEIROS E MATRIZES

Por definição, o valor de uma variável do tipo vetor é o endereço da primeira posição do vetor. Ou seja,

**p1 = x;**

é uma expressão válida. Onde p1 recebe o endereço da primeira posição do vetor x. Mesma coisa de **p1 = &x[0]**

```
int x[10], *p;
x[0]=5;
x[1]=2;
p = x;
printf("%d\n", *p);           //exibe 5
printf("%d\n", p[0]);         //exibe 5
printf("%d\n", *(++p));      //exibe 2
printf("%d\n", *x);           // mesma coisa de x[0]. Exibe 5
printf("%d\n", *(x+1));       // mesma coisa de x[1]. Exibe 2
```

```
char nome[30] = "José da Silva";
char *p1, *p2;
char car;
int i;
p1 = nome;                                     // nome sozinho é um ponteiro para o 1º
                                              // elemento de nome[].

car = nome[3];                                // Atribui 'é' a car.
car = p1[0];                                  // Atribui 'J' a car. Válido.
p2 = &nome[5];                                // Atribui a p2 o endereço da 6ª
                                              // posição de nome, no caso 'd'.

printf( "%s", p2);                             // Imprime "da Silva"...

p2 = p1;                                       // Evidentemente válido.
p2 = p1 + 5;                                  // Equivalente a p2 = &nome[5]
printf( "%s", (p1 + 5));                      // Imprime "da Silva"...
printf( "%s", (p1 + 20));                     // Cuidado: Imprime lixo!!

for (i=0; i<=strlen(nome)- 1; i++)
{
    printf ("%c", nome[i]);                  // Imprime 'J','o','s',etc
    p2 = p1 + i;
    printf ("%c", *p2);                      // Imprime 'J','o','s',etc
}
```

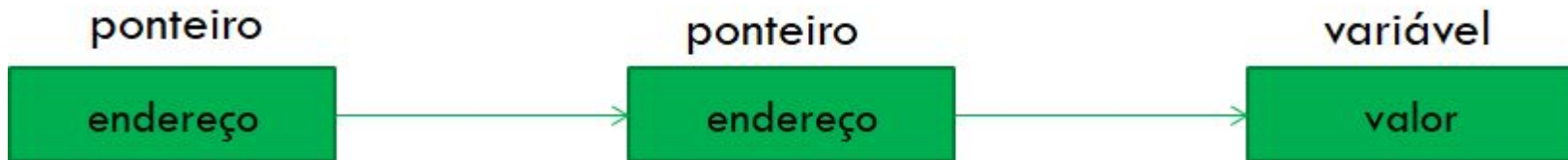
# INDIREÇÃO MÚLTIPLA

- ❑ É possível ter um ponteiro apontando para outro ponteiro que aponta para o valor final

indireção simples



indireção múltipla





# INDIREÇÃO MÚLTIPLA

```
#include <stdio.h>
main (){
    int x, *p, **q;                                /* q é um ponteiro para um ponteiro de
                                                    inteiro*/

    x = 10;
    p = &x;                                         /* p aponta para x */
    q = &p;                                         /* q aponta para p */
    printf ("%i\n", **q);                          /* imprime 10... */
}
```

# PASSAGEM DE PARÂMETROS

- ❑ Por valor

- ❑ Quando copiamos o valor de uma variável para dentro do parâmetro de uma função

- ❑ Por referência

- ❑ Quando passamos para uma função uma referência a uma região de memória onde está o valor desta variável

# PASSAGEM POR REFERÊNCIA

- ❑ Em C todas as chamadas de funções são feitas por valor;
- ❑ Através do return é possível retornar um valor de uma função ou retornar o controle a função que a chamou;
- ❑ Às vezes há necessidade de alterar uma ou mais variáveis no local que chamou a função;
- ❑ Os ponteiros junto com o operador de referência indireta (\*) permitem chamadas por referência;
- ❑ Quando se deseja trabalhar com ponteiros os argumentos são normalmente passados com o operador (&).

# PASSAGEM DE PARÂMETRO

- ❑ Quando realizamos a passagem de vetores para uma função, o que é passado é a **posição do elemento inicial**, ou seja, a passagem é feita **por referência**;

```
void display (int num[10]);  
void display (int num[]);  
void display (int *num);
```

```
void display ( ) {  
    int i;  
    for (i=0; i<10; i++){  
        printf ("%d ", num[i]);  
    }  
}
```

# EXEMPLO

```
void troca(int *x, int *y);
```

```
int main (){  
    int a, b;  
    a = 10;  
    b = 20;  
    troca (&a, &b);  
}
```

```
void troca (int *x, int *y) {  
    int temp;  
    temp = *x; /* salva o valor contido no endereço x */  
    *x = *y      /* põe valor de y em x */  
    *y = temp; /* põe valor de x em y */  
}
```

# EXERCÍCIOS

1. Declare um vetor do tipo float chamado números com 10 posições e os inicialize com os números 0.0, 1.1, 2.2, 3.3, ..., 9.9. Declare um ponteiro nPtr do tipo float. Faça um laço for para imprimir todas as posições do array números. Percorra o array imprimindo cada posição através do ponteiro nPtr.
  - a.
2. Escrever um programa para ler uma frase qualquer do teclado e imprimir, esta mesma frase, um caractere por vez usando ponteiro.

# ALOCAÇÃO DINÂMICA DE MEMÓRIA

# ALOCACÃO DE MEMÓRIA

- ❑ Uso da memória em C
  - ❑ uso de variáveis globais (e estáticas). O espaço reservado para uma variável global existe enquanto o programa estiver sendo executado;
  - ❑ uso de variáveis locais. Neste caso, o espaço existe apenas enquanto a função que declarou a variável está sendo executada;
  - ❑ reservar memória requisitando ao sistema, em tempo de execução, um espaço de um determinado tamanho – Alocação Dinâmica de Memória



# ALOCACÃO DINÂMICA DE MEMÓRIA

- ❑ Muitas vezes a quantidade de memória a se alocar só é conhecida em tempo de execução;
- ❑ Além disso, definir um tamanho máximo para suas estruturas de dados gera desperdício de memória;
- ❑ A solução é alocar a memória necessária, quando realmente precisar dela – Alocação dinâmica
- ❑ Exemplo:
  - ❑ Processador de texto
  - ❑ Banco de dados

# ALOCACÃO DINÂMICA DE MEMÓRIA

- ❑ O espaço alocado dinamicamente permanece reservado até que seja explicitamente liberado pelo programa.
- ❑ A partir do momento que liberarmos o espaço, ele fica disponível para outros usos e não podemos mais acessá-lo.

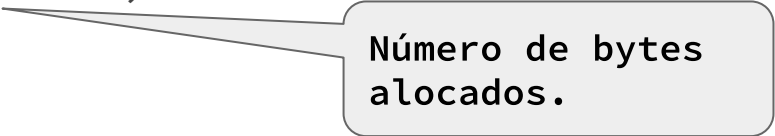
# INSTRUÇÕES

- ❑ C define 4 instruções para gerenciar alocação dinâmica de memória. Disponíveis na biblioteca `<stdlib.h>`
- ❑ São elas:
  - ❑ `malloc`
  - ❑ `calloc`
  - ❑ `realloc`
  - ❑ `free`

# INSTRUÇÕES

- ❑ malloc – permite alocar blocos de memória em tempo de execução

```
void *malloc (int tamanho)
```



Número de bytes  
alocados.

- ❑ retorna um ponteiro void para n bytes de memória não iniciados. Se não há memória disponível malloc retorna NULL

# INSTRUÇÕES

calloc

```
void *calloc(n, size);
```

Tamanho em  
bytes de cada  
elemento

Número de  
elementos a  
ser alocado

- ❑ calloc retorna um ponteiro para um array com n elementos de tamanho size cada um ou NULL se não houver memória disponível.

# INSTRUÇÕES

- ❑ Código que aloca memória para um inteiro

```
int *p;
```

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

- ❑ Código que aloca memória para um vetor de 50 inteiros

```
int *ai = (int*) calloc (50, sizeof(int));
```

# INSTRUÇÕES

- ❑ `realloc` – usado para redimensionar o espaço alocado previamente com `malloc` ou `calloc`
- ❑ Exemplo

```
int *A;
A = calloc (n,sizeof(double));
...
RA = realloc (A,2*n);
```

# INSTRUÇÕES

- ❑ Toda memória não utilizada deve ser liberada.  
Para isso usa-se a instrução `free()`.

```
int *p;  
p = (int * )malloc (sizeof(int));  
free (p);
```



# EXERCÍCIOS

3. Faça um programa em C que contenha um vetor de idades com dimensão escolhida pelo usuário. Coloque em cada uma das posições do vetor valores referentes a idades até o valor escolhido pelo usuário. Depois exiba cada um dos valores usando ponteiros. Use alocação dinâmica.
4. Faça um programa em C para alocar espaço dinamicamente para colocar o nome do usuário (30 caracteres). Se conseguir alocar, leia esta string e mostre-a ao contrário.

# REFERÊNCIAS

Consultar ementário.