



Estruturas de Informação Aula 4: Ponteiros

24/03/2008

Ponteiros

- Permite o armazenamento e manipulação de endereços de memória
- *Forma geral de declaração*
 - *tipo *nome ou tipo* nome*
 - Símbolo * indica ao compilador que a variável guardará o endereço da memória
 - Neste endereço da memória haverá um valor do tipo especificado (tipo_do_ponteiro)
 - `char *p;` (p pode armazenar endereço de memória em que existe um carácter armazenado)
 - `int *v;` (v pode armazenar endereço de memória em que existe um inteiro armazenado)
 - `void *q;` (ponteiro genérico)

Recap: Ponteiros (2)

- Exemplo

```
/*variável inteiro*/
int a;

/*variavel ponteiro para inteiro */
int* p;

/* a recebe o valor 5*/
a = 5;

/* p recebe o endereço de a */
p = &a;

/*conteúdo de p recebe o valor 6 */
*p = 6;
```

Recap: Ponteiros (3)

- Exemplo

```
int main (void)
{
    int a;
    int *p;
    p = &a;
    *p = 2;
    printf (" %d ", a)
    return 0;
}
```

Recap: Ponteiros (4)

- Exemplo

```
int main (void)
{
    int a, b, *p;
    a = 2;
    *p = 3;
    b = a + (*p);
    printf (" %d ", b);
    return 0;
}
```

Declarações que também são ponteiros

- Nome (sozinho) é um ponteiro para carácter que aponta para o primeiro elemento do nome;

```
int v[20], *p;

p = &v[5];
*p = 0;          /* equivante a fazer v[5] = 0
```

```
char nome[30];
char *apontaPraNome;
...
apontaPraNome = nome; /* só o endereço */
```

Operadores de ponteiros



- * indireção
 - Devolve o valor apontado pelo ponteiro
- & operador de endereço
 - Devolve o endereço na memória de seu operador
- main ()

```
{
    int *aponta;
    int valor1, valor2;
    valor1 = 5;
    aponta = &valor1;
    valor2 = *aponta;
}
```
- Precedência: operadores & e * têm precedência maior que outros operadores (com exceção do menos unário)
 - int valor; int *aponta; valor = *aponta++

Aritmética de ponteiros (1)



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

```
int *p1, *p2, x;
x = 4;
p1 = &x;
p2 = p1;
```

Aritmética de ponteiros (2)



- Adição e subtração

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

- Neste exemplo, p2 e p3 apontam para endereços de memória que não estão associados com nenhuma variável. Neste caso, expressões do tipo *p2 e *p3 resultam em ERRO. O único endereço de memória acessável é o de x.

Aritmética de ponteiros (3)



- Importante!
 - As operações de soma e subtração são baseadas no tamanho do tipo base do ponteiro
 - Ex.: se p1 aponta para 2000, p1 + 2 vai apontar para:
 - 2002, se o tipo base do ponteiro for char (1 byte)
 - 2008, se o tipo base do ponteiro for int (4 bytes)
 - Ou seja, este exemplo de soma significa que o valor de p1 é adicionado de duas vezes o tamanho do tipo base.

Aritmética de ponteiros (4)



- No exemplo anterior, se x = 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 o tipo base dos ponteiros acima fosse char* (1 byte), os endereços seriam, respectivamente: 1000, 1001, 1001, 1005 e 1000.

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

Aritmética de ponteiros (5)



- Explique a diferença entre: (int *p) p++; (*p)++; *(p++);
- Comparação entre ponteiros (verifica se um ponteiro aponta para um endereço de memória maior que outro)

```
int *p; *q;
...
if (p < q)
    printf ("p aponta para um endereço menor que o de q");
```

Ponteiros, Vetores e Matrizes



- Ponteiros, vetores e matrizes são muito relacionados em C
- Já vimos que vetores também são ponteiros.
 - `char nome[30]`
 - `nome` sozinho é um ponteiro para caracter que aponta para a primeira posição do nome
- As seguintes notações são equivalentes:
 - `variável[índice]`
 - `* (variável+índice)`
 - `variável[0]` equivale a `*variável !`

Exemplo



```
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.
p2 = &nome[5]; // Atribui a p2 o endereço da 6ª
               // posição de nome, no caso 'd'.
printf( "%s", p2); // Imprime "da Silva"...

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

Exemplo (2)



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

Matrizes de ponteiros



- Ponteiros podem ser declarados como vetores ou matrizes multidimensionais. Exemplo:

```
int *vetor[30]; /* Vetor de 30 ponteiros
                /* para números inteiros */
int a=1, b=2, c=3;

vetor[0] = &a; /* vetor[0] aponta para a*/
vetor[1] = &b;
vetor[2] = &c;
/* Imprime "a: 1, b: 2"... */
printf ( "a: %i, b: %i", *vetor[0],
        *vetor[1] );
```

Matrizes de ponteiros (2)



- Importante:
 - Quando alocamos um vetor de ponteiros para inteiros, não necessariamente estamos alocando espaço de memória para armazenar os valores inteiros!
- No exemplo anterior, alocamos espaço de memória para a, b e c (3 primeiras posições do vetor apontam para as posições de memória ocupadas por a, b, e c)

Matrizes de ponteiros



- Matrizes de ponteiros são muito utilizadas para manipulação de string. Por exemplo:

```
char *mensagem[] = { /* vetor inicializado */
    "arquivo não encontrado",
    "erro de leitura",
    "erro de escrita",
    "impossível criar arquivo"
};
void escreveMensagemDeErro (int num)
{
    printf ("%s\n", mensagem[num]);
}
main ()
{
    escreveMensagemDeErro ( 3 );
}
```

Callout bubble: %s imprime a string até encontrar o car. "\0"

Matrizes de ponteiros (2)



- Manipular inteiros é um pouco diferente:

```
int *vetor[40];
void imprimeTodos ()
{   int i;
    for (i=0; i < 40; i++)
        printf ("%i\n", *vetor[i]);
}
```

- *vetor[i] equivale a *(vetor +i)
- Vetor aponta para um ponteiro que aponta para o valor do inteiro
- Indireção Múltipla ou de Ponteiros para Ponteiros

Ponteiro para Ponteiros ou Indireção Múltipla



- Podemos usar ponteiros para ponteiros implicitamente, como no exemplo anterior
- Também podemos usar uma notação mais explícita, da seguinte forma:
 - tipo **variável;
- **variável é o conteúdo final da variável apontada;
- * variável é o conteúdo do ponteiro intermediário.

Ponteiro para Ponteiros (2)



```
#include <stdio.h>
main ()
{
    int x, *p, **q;
    x = 10;
    p = &x;           // p aponta para x
    q = &p;           // q aponta para p
    printf ("%i\n", **q); // imprime 10...
}
```

Explique o comportamento do seguinte programa:



```
#include <stdio.h>
char *a = "Bananarama";
char b[80] = "uma coisa boba";
char *c[5];

/* Recebe vetor de ponteiros para caracter de tamanho indefinido */
void teste1 (char *d[])
{
    printf( "Teste1: d[0]:%s e d[1]:%s\n", d[0], d[1]);
}

/* Recebe ponteiro para ponteiro para caracter */
void teste2 (char **d )
{
    printf( "Teste2: d[0]:%s e d[1]:%s\n", d[0], d[1]);
    printf( "Teste3: d[0]:%s e d[1]:%s\n", *d, *(d + 1));
}

main ()
{
    c[0] = a;
    c[1] = b;
    printf( "a: %s e b: %s\n", a, b);
    printf( "c[0]: %s e c[1]: %s\n", c[0], c[1]);
    teste1 ( c );
    teste2 ( c );
}
```



Próxima aula



- Exemplos/exercícios com ponteiros e alocação dinâmica