

# Ponteiros

Cleo Zanella Billa

FURG - Fundação Universidade Federal de Rio Grande  
C3 - Centro de Ciências Computacionais

# Objetivo

Conceituar e ilustrar a aplicação de variáveis do tipo ponteiro, em algoritmos e programação de computadores.

# Motivação

Ponteiros são muito poderosos e permitem acesso direto a memória. Com isso o programador ganha bastante liberdade para otimizar o uso de memória através de estruturas de dados de tamanho variável.

Aplicações diretas: alocação dinâmica de memória e passagem de parâmetros por referência.



# Ponteiros

- ▶ Ponteiros ou apontadores são variáveis que armazenam um endereço de memória.
- ▶ Quando um ponteiro contém o endereço de uma variável, dizemos que o ponteiro está “apontando” para essa variável;



# Declaração

Na linguagem C, deve-se informar que tipo de informação existe no endereço apontado pelo ponteiro.

Ou seja, deve-se declarar um ponteiro para um inteiro, ou um ponteiro para um char, ou ponteiro para um float, ...

Sintaxe:

```
<tipo> * nomePonteiro;
```

Exemplos:

```
int *p;  
float *q;  
char* s;
```

# Atribuindo um endereço a um ponteiro

Supondo que temos uma variável  $V$ , e queremos que o ponteiro  $P$ , aponte para esta variável.

Para isso, usa-se o operador  $\&$  (*AddressOf*) para obter o endereço de  $V$ .

$$P = \&V;$$

*(P recebe o endereço de V)*



# Atribuindo um endereço a um ponteiro

Exemplo:

```
int main() {  
    int x=0,y=1,z=2;  
    int *p; int *q;  
  
    x = 10;  
    p = &x;  
    y = 20;  
    z = 30;  
    q = &z;  
    p = &y;  
  
    return 0;  
}
```

# NULL

Existe um valor especial que serve para representar um endereço de memória nulo. Ele é muito útil para inicializar os ponteiros.

Em C, o endereço nulo é dado pela constante NULL (tudo em maiúsculo). Ex.:

```
P = NULL;
```

# Acessando o conteúdo do que o ponteiro está apontando

- ▶ Para acessar a memória que o ponteiro *p* está apontando, usamos o operador *\**. Ex.: *\*p*
- ▶ Exemplo código:

```
int main() {  
    int x;  
    int *p=NULL;  
  
    x = 10;  
    p = &x;  
    printf("P aponta para o valor: %d \n",*p);  
    *p = 20;  
    printf("P aponta para o valor: %d \n",*p);  
    printf("X: %d \n",x);  
  
    return 0;  
}
```

# Acessando o conteúdo do que o ponteiro está apontando

```
int main() {  
    int x;  
    int *p=NULL;  
  
    x = 10;  
    p = &x;  
    printf("P aponta para o valor: %d \n",*p);  
    /* P aponta para o valor: 10 */  
  
    *p = 20;  
    printf("P aponta para o valor: %d \n",*p);  
    /* P aponta para o valor: 20 */  
  
    printf("X: %d \n",x);  
    /* X: 20 */  
    return 0;  
}
```

# CUIDADO!!

Erros comuns utilizando ponteiros:

- ▶ Ponteiro receber um valor e não um endereço.

```
int x=10;
int *p;
p=x; /* ERRADO - correto: p=&x */
```

- ▶ Alterar o valor de um ponteiro não inicializado.

```
int x=10;
int *p;
*p=20; /* ERRADO - correto: p=&x; *p=20; */
```

- ▶ Sempre que uma variável ou um ponteiro é criado, não se sabe qual o conteúdo desse, por isso é **MUITO IMPORTANTE** inicializar a variável ou o ponteiro antes de usar.

## Exemplo 01

Programa	Memória
int x; int *p, *q;	
x = 10;	
p = &x;	
q = p;	
*q = *p + 10;	
printf ("%d \n", *p);	

# Exemplo 01

Programa	Memória
int x; int *p, *q;	x=? p → ?    q → ?
x = 10;	x=10 p → ?    q → ?
p = &x;	x=10 p → 10    q → ?
q = p;	x=10 p → 10    q → 10
*q = *p + 10;	x=20 p → 20    q → 20
printf ("%d \n", *p);	Imprime o valor 20

## Exemplo 02

```
int main() {  
    int *p,x; int y;  
    p=&x;  
    y=5;  
    x=10;  
    printf(" *p=%d x=%d y=%d \n",*p,x,y);  
    p=&y;  
    y>(*p)+10;  
    printf(" *p=%d x=%d y=%d \n",*p,x,y);  
}
```



## Exemplo 02

```
int main() {  
    int *p,x; int y;  
    p=&x;  
    y=5;  
    x=10;  
    printf(" *p=%d x=%d y=%d \n",*p,x,y);  
    p=&y;  
    y>(*p)+10;  
    printf(" *p=%d x=%d y=%d \n",*p,x,y);  
}
```

R:

```
/* Imprime: *p=10 x=10 y=5 */  
/* Imprime: *p=15 x=10 y=15 */
```

## Exemplo 03

```
int main() {  
    int *p,*q;  
    int x,y,z;  
    x=0;  
    p=&x;  
    y=1;  
    q=&z;  
    *q=2;  
    printf("x=%d y=%d z=%d *p=%d *q=%d \n",x,y,z,*p,*q);  
    *p=(*p)+y+(*q);  
    printf("x=%d y=%d z=%d *p=%d *q=%d \n",x,y,z,*p,*q);  
    p=&z;  
    q=&y;  
    printf("x=%d y=%d z=%d *p=%d *q=%d \n",x,y,z,*p,*q);  
}
```

## Exemplo 03

```
int main() {  
    int *p,*q;  
    int x,y,z;  
    x=0;  
    p=&x;  
    y=1;  
    q=&z;  
    *q=2;  
    printf("x=%d y=%d z=%d *p=%d *q=%d \n",x,y,z,*p,*q);  
    *p=(*p)+y+(*q);  
    printf("x=%d y=%d z=%d *p=%d *q=%d \n",x,y,z,*p,*q);  
    p=&z;  
    q=&y;  
    printf("x=%d y=%d z=%d *p=%d *q=%d \n",x,y,z,*p,*q);  
}
```

R:

```
/* Imprime: x=0 y=1 z=2 *p=0 *q=2 */  
/* Imprime: x=3 y=1 z=2 *p=3 *q=2 */  
/* Imprime: x=3 y=1 z=2 *p=2 *q=1 */
```

## Exemplo 04

```
int main() {  
    int x;  
    int *p;  
    int **pp;  
    x=10;  
    p=&x;  
    pp=&p;  
    printf("x=%d *p=%d **pp=%d \n",x,*p,**pp);  
    (**p)++;  
    printf("x=%d *p=%d **pp=%d \n",x,*p,**pp);  
}
```

## Exemplo 04

```
int main() {  
    int x;  
    int *p;  
    int **pp;  
    x=10;  
    p=&x;  
    pp=&p;  
    printf("x=%d *p=%d **pp=%d \n",x,*p,**pp);  
    (**p)++;  
    printf("x=%d *p=%d **pp=%d \n",x,*p,**pp);  
}
```

```
/* Imprime: x=10 *p=10 **pp=10 */  
/* Imprime: x=11 *p=11 **pp=11 */
```

# Alocação Dinâmica de Memória

- ▶ Usar somente a memória necessária, diminuindo assim a quantidade de recursos necessários para a execução do programa.
- ▶ Não impor limites quanto aos recursos utilizados, a não ser físicos (e.g. tamanho da memória), para os programas.

## malloc()

- ▶ Para alocar dinamicamente memória (em tempo de execução do programa) precisamos usar ponteiros e uma função denominada `malloc()`.
- ▶ `malloc` é a abreviatura de *memory allocation*.
- ▶ A função `malloc()` recebe o tamanho em bytes da quantidade de memória que deve ser alocada e retorna o endereço desse espaço de memória.
- ▶ Toda memória alocada com o `malloc` deve ser liberada. A função `free` libera a memória.
- ▶ `malloc` e `free` pertencem a biblioteca `stdlib.h`

## Exemplo 01

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    char *ptr;

    // Aloca a memoria
    ptr = (char *) malloc (1); // Por que 1 ?
                                // Por que (char *) ?

    scanf ("%c", ptr); // Por que nao tem & ?
    printf ("%c\n", *ptr); // Por que tem * ?

    // Libera a memoria.
    free (ptr);
}
```



## Exemplo 01

► `ptr = (char *) malloc (1); // Por que 1 ?`

## Exemplo 01

► `ptr = (char *) malloc (1); // Por que 1 ?`

Porque um char tem um 1 byte.

## Exemplo 01

- ▶ `ptr = (char *) malloc (1); // Por que 1 ?`

Porque um char tem um 1 byte.

- ▶ `ptr = (char *) malloc (1); // Por que (char *) ?`

# Exemplo 01

- ▶ `ptr = (char *) malloc (1); // Por que 1 ?`

Porque um char tem um 1 byte.

- ▶ `ptr = (char *) malloc (1); // Por que (char *) ?`

Porque `malloc` retorna um endereço de memória puro, precisamos dizer explicitamente qual o tipo de conteúdo tem no endereço de memória retornado. Isso se chama *type cast* ou coerção de tipo.

# Exemplo 01

► `scanf ("%c", ptr); // Por que nao tem & ?`

# Exemplo 01

► `scanf ("%c", ptr); // Por que nao tem & ?`

Porque ptr é um ponteiro. Não queremos o endereço do ponteiro e sim endereço armazenado nele.

## Exemplo 01

- ▶ `scanf ("%c", ptr); // Por que nao tem & ?`

Porque ptr é um ponteiro. Não queremos o endereço do ponteiro e sim endereço armazenado nele.

- ▶ `printf ("%c\n", *ptr); // Por que tem * ?`

## Exemplo 01

- ▶ `scanf ("%c", ptr); // Por que nao tem & ?`

Porque ptr é um ponteiro. Não queremos o endereço do ponteiro e sim endereço armazenado nele.

- ▶ `printf ("%c\n", *ptr); // Por que tem * ?`

Porque ptr é um ponteiro. Queremos imprimir o conteúdo do endereço que ele aponta e não o endereço armazenado nele.



## Exemplo 02

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int *ptr;
    int x;

    // Aloca memoria
    ptr = (int *)malloc (sizeof (int));

    scanf ("%d %d", ptr, &x);
    *ptr = *ptr + x;
    printf ("%d\n", *ptr);

    // Libera a memoria
    free (ptr);
}
```

A expressão `sizeof(int)` dá o número de bytes de um `int`.

# Ponteiros e Vetores

Todo vetor é um ponteiro. Por exemplo:

```
int V[10];  
int *P;  
  
/*P pode receber V, jah que V eh um ponteiro */  
P=V;  
  
/* Posso usar P para acessar os elementos do vetor */  
P[2]=20;  
  
/* Posso usar o notacao de ponteiro para acessar  
   os elementos do vetor */  
printf("%d", *(P+2) );  
  
/* V[0] == *(P+0) */
```

**CUIDADO** - Não se pode alterar o vetor ... algo como `V=&x;`. O vetor funciona como um ponteiro constante.

# Alocando Dinamicamente um Vetor

- ▶ Basta alocar o tamanho em bytes para conter todos os elementos do vetor.
- ▶ Ex.: Um vetor de 100 números inteiros precisa de  $100 * \text{sizeof}(\text{int})$  bytes.
- ▶ Do ponto de vista conceitual o comando

```
v = (int *)malloc(100*sizeof(int));
```

tem o mesmo efeito que:

```
int v[100];
```

# Alocando Dinamicamente um Vetor

- ▶ Exemplo para alocar um vetor de tamanho  $n$  definido pelo usuário.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int *v, n, i;
    // Le o tamanho do vetor
    scanf ("%d", &n);
    // Aloca o vetor de tamanho n
    v = (int *)malloc(n * sizeof(int));
    //Le os elementos do vetor
    for (i = 0; i < n; i += 1)
        scanf ("%d", &v[i]);
    // Mostra os elementos do vetor
    for (i = 0; i < n; i += 1)
        printf ("%d ", v[i]);
    // Libera espaco alocado
    free (v);
}
```

# Memória não é infinita

- ▶ Se a memória do computador já estiver toda ocupada, `malloc()` não consegue alocar mais espaço e devolve `NULL`.
- ▶ O ideal é verificar essa possibilidade antes de prosseguir:

```
ptr = (int *)malloc(sizeof(int)*1000000000);  
if (ptr == NULL) {  
    printf ("Erro!! Memoria insuficiente\n");  
}
```

## calloc()

- ▶ A linguagem C também apresenta a função `calloc()` que é muito parecida com o `malloc()`.
- ▶ A função `calloc()` recebe dois argumentos, o primeiro é o número de células de memória desejado e o segundo é o tamanho de cada célula em bytes. Ex.:

```
int *mem;  
mem = (int *)calloc(100, sizeof(int));
```

- ▶ É equivalente a:  
`mem = (int *)malloc(100*sizeof(int));`
- ▶ Outra diferença do `calloc()` é que ele inicializa todo o conteúdo alocado com **zeros**.

## Exercício 01

Escreva um programa que leia um número inteiro positivo  $n$  seguido de  $n$  números inteiros e imprima esses  $n$  números em ordem invertida. Por exemplo, ao receber

5   222 333 444 555 666

o seu programa deve imprimir

666 555 444 333 222

$n$  pode ser qualquer valor maior ou igual a zero.

# Resposta Exercício 01

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int n, *vetor;

    printf("Entre com o numero de elementos: ");
    scanf ("%d", &n);

    /* Alocando espaco para n inteiros */
    vetor = (int *) malloc(n * sizeof (int));

    for (i = 0; i < n; i += 1) {
        printf(" Digite elemento %d: ", i);
        scanf ("%d", &vetor[i]);
    }
    printf(" Imprimindo na ordem inversa: \n");
    for (i = n-1; i >=0; i -= 1)
        printf ("%d \n", vetor[i]);

    /* Nao esquecer de liberar o espaco alocado */
    free (v);
}
```



# Referências

- ▶ ASCENCIO, Ana F. G.; CAMPOS, Edilene A. V. Fundamentos da programa de computadores. Pearson Prentice Hall, 2007.
- ▶ Schildt, Herbert. C completo e total. Pearson Makron Books, 1997.
- ▶ Notas de aula do Prof. Flavio Keidi Miyazawa.
- ▶ Notas de aula do Prof. Emanuel Estrada.
- ▶ Notas de aula do Prof. Alessandro Bicho.