

INSTITUTO FEDERAL DE  
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA  
BAHIA



# **Instituto Federal da Bahia**

## **Análise e Desenvolvimento de Sistemas**

### **INF029 – Laboratório de Programação**

## **Aula 03: Ponteiros**

Prof. Dr. Renato L. Novais  
[renato@ifba.edu.br](mailto:renato@ifba.edu.br)

# Agenda



- Ponteiros

# Roteiro



- Ponteiros
  - Definição.
  - Operadores.
  - Ponteiros e Variáveis.
  - Ponteiros e Vetores.
  - Ponteiros e Funções.
- Dúvidas
- Exercícios

# Ponteiros - Definição



- Ponteiros são tipos de dados que **referenciam** (ou “apontam” para) **endereços de memória**.
- Em algumas linguagens com **maior abstração**, **ponteiros não existem expostos ao programador** (como em **Java**) ou tem alternativas mais seguras em outros tipos de dados (como o tipo “referência” em C++).
- Acessar o valor nesse endereço é chamado de “**dereferenciar**” o ponteiro

# Ponteiros - Definição

- Em C, **um ponteiro é um número inteiro**, referindo-se ao endereço na memória.
- Com ponteiros é possível:
  - **Funções modificar seus argumentos**
  - Fazer rotinas de **alocação dinâmica**
  - Aumentar a **eficiência** de certas rotinas

# Ponteiros - Definição



- A sintaxe para declarar um ponteiro, em C, é:  
`tipo *ponteiroDeTipo;`
- Como qualquer outro tipo, podemos ter vetores de ponteiros:  
`tipo *vetorDePonteiros[tamanho];`
- Tecnicamente um ponteiro poderia ser de qualquer tipo, uma vez que guarda o endereço de memória
  - Entretanto tem-se a Aritmética de ponteiros
- E, como um ponteiro é um tipo, podemos ter ponteiros para ponteiros (ad infinitum):

`tipo **ponteiroDePonteiroDeTipo;`

`tipo ***ponteiroDePonteiroDePonteiroDeTipo;`

# Ponteiros – Operadores



- Para a atribuição de valores para ponteiros, usamos o operador "=", como fizemos com qualquer outro tipo, MAS:
  - Mesmo sendo um inteiro, **não se atribui (normalmente\*) valores arbitrários a ponteiros, pois são raras as ocasiões em que é necessário usar um endereço constante para todas as execuções do programa.**
  - Para contornar isso, precisamos saber o **endereço de variáveis em tempo de execução**. Conseguimos isso através do operador "&" ("endereço de"):

```
int inteiro;  
int* pInt = &inteiro;
```

- Como vetores já são endereços, podemos usar a atribuição sem precisar achar o endereço com o "&":

```
int vetorInt[5];  
int* pVetor = vetorInt;
```

\*: Exceção para o endereço NULL, que equivale ao endereço 0, normalmente retornado por funções em caso de erro, ou para indicar que o ponteiro não aponta para lugar nenhum.

## Exemplo



```
#include <stdio.h>
#include <stdlib.h>
```

```
int ponteirosStart()
{
```

```
    printf("Hello ponteiros!\n");
```

```
    int i = 2;
```

```
    int *p = &i;
```

```
    printf("Endereço de i: %p\n", p);
```

```
    printf("Valor de i %d\n", *p);
```

```
    //printf("%p\n", &ponteiro);
```

```
    return 0;
```

```
}
```

Hello ponteiros!

Endereço de i: 0x7fff57b63abc

Valor de i 2

## Exemplo



- Faça um programa que tenha o seguinte código fonte
- Coloque a e b como duas variáveis informadas pelo usuário
- Imprima o valor de c

```
int *p;          /* p é um ponteiro para um inteiro */
int *q;
p = &a;          /* o valor de p é o endereço de a */
q = &b;          /* q aponta para b */
c = *p + *q;
```

# Ponteiros – Operadores



- Para o **acesso ao conteúdo** no endereço ao qual o ponteiro aponta, usamos o **operador "\*" (dereferenciador)**:

```
int* pInt;  
*pInt = 2;  
int inteiro = *pInt;
```

- Podemos também usar a notação de vetores para acessar o conteúdo de um ponteiro (**pois ponteiros são endereços, e vetores também\***):

```
int vetorInt[6];  
int* pVetor = vetorInt;  
int inteiro = pVetor[5];
```

- Que é equivalente a:

```
inteiro = *(pVetor + 5);
```

\*: da mesma forma, podemos usar o operador de dereferencia em vetores.

# Ponteiros – Operadores



- Porém, ao dereferenciar ou acessar um ponteiro, devemos ter cuidado: um ponteiro com um endereço de memória inválido ou nulo, ao ser dereferenciado, irá causar um erro de “Falha de segmentação” (segmentation fault), finalizando forçadamente a execução de seu programa.
- Coisas como essa precisam ser evitadas:

```
double* pDouble;  
*pDouble = 2.5;
```

```
long* pLong;  
int inteiro = *pLong;
```

```
char* string = NULL;  
puts(string);
```

# Ponteiros – Operadores



- Então podemos pensar, corretamente, que ao declarar uma variável da forma

```
float *pFloat;
```

Estamos declarando que o *conteúdo* ao qual pFloat aponta é do tipo *float*, tornando pFloat um ponteiro para float.

- Por isso, a seguinte declaração também é válida:

```
char (*pString)[50];
```

Declarando que o conteúdo ao qual pString aponta é do tipo vetor de char de 50 posições, tornando pString um ponteiro para vetor de char de 50 posições.

- Como estamos declarando um conteúdo, e não a variável em si, a memória não é reservada para essa variável, que inicialmente aponta para um endereço qualquer na memória.

# Ponteiros – Operadores



- Podemos usar as operações de adição e subtração com ponteiros. Isso permite coisas desse tipo:

```
pInt = (pVetor + 5);  
pVetor++;  
pVetor--;  
pChar -= 3;
```

- Multiplicação e divisão não são suportadas, nem soma de dois ponteiros, pois isso não faz sentido se tratando de memória.
- As operações de adição, subtração, incremento e decremento se dão em função do tamanho do tipo para o qual o ponteiro aponta. Se tivermos um ponteiro para inteiro e incrementarmos esse ponteiro por um, ele apontará para o endereço de memória 4 bytes adiante.

# Ponteiros e Variáveis



- Podemos, usando os operadores apresentados, fazer um ponteiro apontar para um endereço o qual armazena uma variável.
- Com isso, podemos **usar o ponteiro para modificar o valor de uma variável:**

```
int inteiro = 0; //inteiro = 0
int* pInt = &inteiro; //pInt aponta para
                        // o endereço de inteiro
*pInt = 42;           //inteiro = 42
```

# Ponteiros e Funções



- Ponteiros, por serem endereços, permitem que acessemos e modifiquemos dados externos à função, de dentro da função, contornando a passagem de variáveis por cópia\*:

```
void copiar(int* a, int b) {  
    *a = b;  
}  
  
int main() {  
    int foo = 2;  
    int bar = 5;  
    //o endereço de foo é passado  
    //como parametro da função  
    copiar(&foo, bar);  
    //foo == 5  
    return 0;  
}
```

\*: A passagem ainda é por cópia, mas o valor copiado é o endereço.

# Ponteiros e Funções



- Podemos, dessa forma, “retornar” mais de um valor por execução de função.
- Isso é muito útil quando é preferível retornar o estado da execução da função, como um código de erro ou de execução correta:

```
int divisao(int* result, float a, float b){  
    int ret = 0;  
    if(b != 0){ //caso tudo ocorra bem,  
                //retornará 0, e *result será a divisão.  
        *result = a/b;  
    }  
    else{ //caso tente dividir por 0,  
          //retorna um código de erro (-1).  
        ret = -1;  
    }  
    return ret;  
}
```

# Exemplos



- Faça um programa que tenha uma função que troca o valor de duas variáveis.
- Quais funções no programa loja poderiam/deveriam usar ponteiros?
- Fazer um programa que tenha duas pilhas de ponteiros (par e impar
  - Adicionar elementos pares e ímpares
  - Remover elementos
  - Listar elementos

# Exemplos



- Escrever uma função (bem como um programa que exercite tal função) que tem como parâmetros três valores inteiros  $a$ ,  $b$  e  $c$  e retorna a posição do maior e a posição do menor valor. Exemplo: Se  $a = 7$ ,  $b = 1$  e  $c = 5$ , o procedimento deve retornar 2 como a posição do menor e 1 como a posição do maior.
- Faça uma função que receba um valor inteiro como referência e retorne o resto da divisão deste número por 10. Altere também o valor da variável passada por referência, dividindo-a por 10.
- Faça um programa que imprima invertido os nomes do algarismos de um número inteiro. (Use a sua função!)  
Ex: 234 saída: quatro três dois

# Ponteiros e Funções



- Outra utilidade da passagem por referência, é quando o resultado precisa ser armazenado em um vetor:
  - Se usássemos uma variável local para o vetor e a retornássemos, ele seria destruído quando a função (seu escopo) terminasse sua execução.
  - Usamos, então, um endereço de um vetor declarado fora da função, que portanto não seria destruído com o término da função:

```
void pegaDiagonal(int* diagonal, int (*matriz)[10], int tam) {  
    int i = 0;  
    for(i = 0; i < tam ; i++)  
        diagonal[i] = matriz[i][i];  
}  
  
int main()  
{  
    int matriz[10][10], diagonal[10];  
    pegaDiagonal(diagonal, matriz, 10);  
    return 0;  
}
```

# Ponteiros e Funções



- Como ponteiros apontam para endereços, eles também podem ser usados para manipular funções:
  - Uma função é uma variável do tipo (tipoRetorno)(tiposParametros). Sua declaração se dá por:  
tipoRetorno nome (tiposParametros)
  - Então se declararmos, usando o operador “\*”, o conteúdo de uma variável como uma função, essa variável será um ponteiro para uma função:

```
int soma(int a, int b) {  
    return a+b;  
}
```

```
int (*func)(int, int);  
func = soma;  
func(a, b);
```

# Ponteiros e Funções



- Então, o código abaixo,

```
int expoente(int a,int b){
    int i;
    for(i = 0; i < b ; i++)
        a *= b;
    return a;
}

int main(){
    int (*funcao)(int,int);
    funcao = expoente;
    printf("%d %d", funcao(2,2), expoente(2,2));
    return 0;
}
```

Terá saída

```
8 8
Process returned 0 (0x0)
```

# Ponteiros e Vetores



- Como já foi dito, ponteiros guardam endereços. Vetores também. Podemos acessar os valores do vetor usando ponteiros? Sim:

```
char string[20];  
char* pChar = string;
```

- Para acessar cada elemento:

```
for(i = 0; i < 20; i++)  
{ //Notação de vetor  
  aux = pChar[i];  
  //Notação de ponteiro  
  *(pChar + i) = funcao();  
}
```

# Ponteiros e Vetores



- Porém, para acessar matrizes através de ponteiros, temos que ter cuidado:
  - Uma matriz é um espaço contínuo na memória, sendo acessado dereferenciando somente um endereço:

```
int matriz[20][10];
```

```
matriz[i][j]; //isso  
*(matriz + i*10 + j); //equivale a isso
```

- Usando ponteiros, um acesso da mesma forma precisa dereferenciar dois endereços:

```
int** matriz;
```

```
matriz[i][j]; //isso  
*(* (matriz + i) + j); //equivale a isso
```

# Ponteiros e Estruturas



```
/* declara uma estrutura */
struct facil {
    int num;
    char ch;
};

main()
{
    /* definioes de variaveis */
    struct facil fac, /* uma variavel do tipo "struct facil" */
                *pfac; /* um ponteiro para "struct facil" */
    pfac = &fac;
    (*pfac).num = 32; /* o membro "num" da "struct facil" apontada por "pfac" */
    (*pfac).ch = 'A'; /* o membro "char" da "struct facil" apontada por "pfac" */
}
```

```
pfac->num = 32; /* o mesmo que (*pfac).num = 32; */
pfac->ch = 'A'; /* o mesmo que (*pfac).ch = 'A'; */
```

## Exercício



- Fazer uma função que ler uma data (mes, dia e ano), usando ponteiro para struct.

```
typedef struct data Data;  
  
struct data {  
    short dia;  
    short mes;  
    int ano;  
};
```

## Exercício



- Faça um programa que contenha duas pilhas uma de elementos pares e uma de ímpares
  - Nenhum parâmetro deve ser público
  - Deve ter uma função para inserir elementos, na pilha par ou ímpar, de acordo com o valor digitado pelo usuário
  - Remover elementos (sempre do final)
  - Listar elementos
  - A função main deve ter um menu com as opções para usuário

# Porque inicializar ponteiros?



- Observe o código:

```
main ()      /* Errado - Nao Execute */
{
    int x, *p;
    x=13;
    *p=x; //posição de memória de p é indefinida!
}
```

- A não inicialização de ponteiros pode fazer com que ele esteja alocando um espaço de memória utilizado, por exemplo, pelo S.O.

## Porque inicializar ponteiros?



- No caso de vetores, é necessário sempre alocar a memória necessária para compor as posições do vetor.
- O exemplo abaixo apresenta um programa que compila, porém poderá ocasionar sérios problemas na execução. Como por exemplo utilizar um espaço de memória alocado para outra aplicação.

```
main() {  
    char *pc; char str[] = "Uma string";  
    strcpy(pc, str); // pc indefinido  
}
```

# Alocação dinâmica de memória



- Durante a execução de um programa é possível alocar uma certa quantidade de memória para conter dados do programa
- A função `malloc (n)` aloca dinamicamente n bytes e devolve um ponteiro para o início da memória alocada
- A função `free(p)` libera a região de memória apontada por p.
  - O tamanho liberado está implícito, isto é, é igual ao que foi alocado anteriormente por malloc.

- Os comandos abaixo **alocam dinamicamente** um inteiro e depois o **liberam**:

```
#include <stdlib.h>
int *pi;
pi = (int *) malloc (sizeof(int));
...
free(pi);
```

- A função malloc não tem um tipo específico (retorna void \*).
  - Assim, (int \*) converte seu valor em ponteiro para inteiro. Como não sabemos necessariamente o comprimento de um inteiro (2 ou 4 bytes dependendo do compilador), usamos como parâmetro a função **sizeof(int)**.

# Alocação dinâmica de vetores



```
#include <stdlib.h>
main() {
    int *v, i, n;
    scanf("%d", &n); // le n
    //aloca n elementos para v
    v = (int *) malloc(n*sizeof(int));
    // zera o vetor v com n elementos
    for (i = 0; i < n; i++) v[i] = 0;
    ...
    // libera os n elementos de v
    free(v);
}
```

# Alocação dinâmica



- Se não houver memória suficiente para alocar a memória requisitada a função malloc() retorna um ponteiro nulo

```
#include <stdio.h>
#include <stdlib.h>      /* Para usar malloc() */
main (void)
{
    int *p;
    int a;
    int i;

    ... /* Determina o valor de a em algum lugar */

    p= malloc(a*sizeof(int));           /* Aloca a números inteiros
                                         p pode agora ser tratado como um vetor com
                                         a posicoes                               */

    if (!p)
    {
        printf ("** Erro: Memoria Insuficiente **");
        exit;
    }

    for (i=0; i<a ; i++)                /* p pode ser tratado como um vetor com a posicoes */
        p[i] = i*i;

    ...

    return 0;
}
```

## realloc()



- `void *realloc (void *ptr, unsigned int num);`
- A função modifica o tamanho da memória previamente alocada apontada por `*ptr` para aquele especificado por `num`.
- O valor de `num` pode ser maior ou menor que o original.
- Um ponteiro para o bloco é devolvido porque `realloc()` pode precisar mover o bloco para aumentar seu tamanho.
  - Se isso ocorrer, o conteúdo do bloco antigo é copiado no novo bloco, e nenhuma informação é perdida.
- Se `ptr` for nulo, aloca `num` bytes e devolve um ponteiro;
- se `num` é zero, a memória apontada por `ptr` é liberada. Se não houver memória suficiente para a alocação, um ponteiro nulo é devolvido e o bloco original é deixado inalterado.

```

#include <stdio.h>

#include <stdlib.h>      /* Para usar malloc() e realloc*/

main (void)
{
    int *p;
    int a;
    int i;

... /* Determina o valor de a em algum lugar */

    a = 30;

    p= malloc(a*sizeof(int));          /* Aloca a números inteiros
                                        p pode agora ser tratado como um vetor com
                                        a posicoes                               */

    if (!p)
    {
        printf ("** Erro: Memoria Insuficiente **");
        exit;
    }

    for (i=0; i<a ; i++)                /* p pode ser tratado como um vetor com a posicoes */
        p[i] = i*i;

    /* O tamanho de p deve ser modificado, por algum motivo ... */

    a = 100;

    p = realloc (p, a*sizeof(int));

    for (i=0; i<a ; i++)                /* p pode ser tratado como um vetor com a posicoes */
        p[i] = a*i*(i-6);

...

    return 0;
}

```

## Exercício



- Crie um programa para alocar espaço para um vetor de inteiros com tamanho definido pelo usuários

## Exercício



- Considere um vetor de ponteiros de 10 posições
  - `int *vetor[10]`
  - Cada ponteiro vai apontar para um vetor de tamanho definido pelo usuário
- Crie uma função `insere` que deve solicitar ao usuário em qual vetor (1 a 10) deve inserir e o número a ser inserido
  - Se for a primeira vez a inserir um elemento no vetor da posição, deve solicitar ao usuário o tamanho do novo vetor e inserir.
- Crie uma função para liberar espaços da memória, o usuário deve informar qual vetor ele quer liberar
- Crie uma função para listar os vetores, se o vetor estiver nulo, imprimir “vetor X vazio”
- Crie uma função que imprima todos os números de todos os vetores ordenados.
  - Deve ser criado um vetor auxiliar do tamanho de todos os outros. Ordene o vetor auxiliar, imprima, e depois libere o espaço de memória desse vetor auxiliar

- Estender o programa anterior, da seguinte forma:
  - Ordene todos os vetores separados, junte-os em uma estrutura auxiliar temporária, imprima ele.
  - A ordenação não deve mudar os dados originais.
  - Você deve liberar qualquer espaço de memória que tenha sido criado temporariamente

## Exercício 3



- Faça um programa que tenha duas matrizes de ordem 3 e dois ponteiros que são inicializados com os endereços do começo das matrizes, use o ponteiro para preencher as matrizes, uma começando pelas linhas e a outra pelas colunas, depois calcule o resultado da soma de matriz em uma terceira matriz usando um terceiro ponteiro para ela (neste ultimo caso, pode-se começar pelas linhas ou colunas, fica a seu critério).

# Ponteiros

```
int ponteirosStart()
{
    printf("Hello ponteiros!\n");
    int i;
    i = 2;
    int *p;
    p = NULL;
    p = &i;

    printf("Endereco de p: %p\n", &p);
    printf("Endereco de i: %p\n", p);
    printf("Valor de i %d\n", *p);
    //printf("%p\n", &pinteiro);
    return 0;
}
```

|   | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| A |   |   |   |   |   |   |
| B |   |   |   |   |   |   |
| C |   |   |   |   |   |   |
| D |   |   |   |   |   |   |
| E |   |   |   |   |   |   |
| F |   |   |   |   |   |   |

# Ponteiros

```
int ponteirosStart()
{
    printf("Hello ponteiros!\n");
    int i;
    i = 2;
    int *p;
    p = NULL;
    p = &i;

    printf("Endereco de p: %p\n", &p);
    printf("Endereco de i: %p\n", p);
    printf("Valor de i %d\n", *p);
    //printf("%p\n", &pinteiro);
    return 0;
}
```

|   | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| A |   |   |   |   |   |   |
| B |   |   |   |   |   |   |
| C |   |   |   |   |   |   |
| D |   |   |   |   |   |   |
| E |   |   |   |   |   |   |
| F |   |   |   |   |   |   |

# Ponteiros

```
int ponteirosStart()
{
    printf("Hello ponteiros!\n");
    int i;
    i = 2;
    int *p;
    p = NULL;
    p = &i;

    printf("Endereco de p: %p\n", &p);
    printf("Endereco de i: %p\n", p);
    printf("Valor de i %d\n", *p);
    //printf("%p\n", &pinteiro);
    return 0;
}
```

|   | 1     | 2 | 3 | 4 | 5 | 6 |
|---|-------|---|---|---|---|---|
| A | I (0) |   |   |   |   |   |
| B |       |   |   |   |   |   |
| C |       |   |   |   |   |   |
| D |       |   |   |   |   |   |
| E |       |   |   |   |   |   |
| F |       |   |   |   |   |   |

# Ponteiros

```
int ponteirosStart()
{
    printf("Hello ponteiros!\n");
    int i;
    i = 2;
    int *p;
    p = NULL;
    p = &i;

    printf("Endereco de p: %p\n", &p);
    printf("Endereco de i: %p\n", p);
    printf("Valor de i %d\n", *p);
    //printf("%p\n", &pinteiro);
    return 0;
}
```

|   | 1     | 2 | 3 | 4 | 5 | 6 |
|---|-------|---|---|---|---|---|
| A | I (2) |   |   |   |   |   |
| B |       |   |   |   |   |   |
| C |       |   |   |   |   |   |
| D |       |   |   |   |   |   |
| E |       |   |   |   |   |   |
| F |       |   |   |   |   |   |

# Ponteiros

```
int ponteirosStart()
{
    printf("Hello ponteiros!\n");
    int i;
    i = 2;
    int *p;
    p = NULL;
    p = &i;

    printf("Endereco de p: %p\n", &p);
    printf("Endereco de i: %p\n", p);
    printf("Valor de i %d\n", *p);
    //printf("%p\n", &ponteiro);
    return 0;
}
```

@ = lixo. Não se sabe para onde o ponteiro está apontando. Pode ser que o compilador aponte para NULL, mas não tem nenhuma garantia disso.

|   | 1     | 2          | 3 | 4 | 5 | 6 |
|---|-------|------------|---|---|---|---|
| A | I (2) | p (@/NULL) |   |   |   |   |
| B |       |            |   |   |   |   |
| C |       |            |   |   |   |   |
| D |       |            |   |   |   |   |
| E |       |            |   |   |   |   |
| F |       |            |   |   |   |   |

# Ponteiros

```

int ponteirosStart()
{
    printf("Hello ponteiros!\n");
    int i;
    i = 2;
    int *p;
    p = NULL;
    p = &i;

    printf("Endereco de p: %p\n", &p);
    printf("Endereco de i: %p\n", p);
    printf("Valor de i %d\n", *p);
    //printf("%p\n", &ponteiro);
    return 0;
}

```

NULL, endereço zero da memória. Usado para inicializar ponteiros

|   | 1     | 2           | 3 | 4 | 5 | 6 |      |
|---|-------|-------------|---|---|---|---|------|
| A | i (2) | p (NULL(0)) |   |   |   |   | *p = |
| B |       |             |   |   |   |   |      |
| C |       |             |   |   |   |   |      |
| D |       |             |   |   |   |   |      |
| E |       |             |   |   |   |   |      |
| F |       |             |   |   |   |   |      |

# Ponteiros

```
int ponteirosStart()
{
    printf("Hello ponteiros!\n");
    int i;
    i = 2;
    int *p;
    p = NULL;
    p = &i;

    printf("Endereco de p: %p\n", &p);
    printf("Endereco de i: %p\n", p);
    printf("Valor de i %d\n", *p);
    //printf("%p\n", &pinteiro);
    return 0;
}
```

|   | 1     | 2      | 3 | 4 | 5 | 6 |         |
|---|-------|--------|---|---|---|---|---------|
| A | i (2) | p (A1) |   |   |   |   | *p = 2  |
| B |       |        |   |   |   |   | &p = A2 |
| C |       |        |   |   |   |   | &i = A1 |
| D |       |        |   |   |   |   | *p = 2  |
| E |       |        |   |   |   |   | p = A1  |
| F |       |        |   |   |   |   |         |

# Ponteiros (com passagens por "referência")

```
int ponteirosStart()
{
    printf("Hello ponteiros!\n");
    int i;
    i = 2;
    int *p;
    p = NULL;
    p = &i;
    funcao(p); //funcao(&i);
    printf("Endereco de p: %p\n", &p);
    printf("Endereco de i: %p\n", p);
    printf("Valor de i %d\n", *p);
    //printf("%p\n", &pinteiro);
    return 0;
}

void funcao(int *j){
    *j = 7;
}
```

|   | 1     | 2      | 3 | 4 | 5 | 6 |         |
|---|-------|--------|---|---|---|---|---------|
| A | i (2) | p (A1) |   |   |   |   | *p = 2  |
| B |       |        |   |   |   |   | &p = A2 |
| C |       |        |   |   |   |   | &i = A1 |
| D |       |        |   |   |   |   | *p = 2  |
| E |       |        |   |   |   |   | p = A1  |
| F |       |        |   |   |   |   |         |

# Ponteiros (com passagens por "referência")

```
int ponteirosStart()
{
    printf("Hello ponteiros!\n");
    int i;
    i = 2;
    int *p;
    p = NULL;
    p = &i;
    funcao(p); //funcao(&i);
    printf("Endereco de p: %p\n", &p);
    printf("Endereco de i: %p\n", p);
    printf("Valor de i %d\n", *p);
    //printf("%p\n", &pinteiro);
    return 0;
}

void funcao(int *j){
    *j = 7;
}
```

|   | 1     | 2      | 3      | 4 | 5 | 6 |         |
|---|-------|--------|--------|---|---|---|---------|
| A | i (2) | p (A1) | j (A1) |   |   |   | *p = 2  |
| B |       |        |        |   |   |   | &p = A2 |
| C |       |        |        |   |   |   | &i = A1 |
| D |       |        |        |   |   |   | p = A1  |
| E |       |        |        |   |   |   | j = A1  |
| F |       |        |        |   |   |   | *j = 2  |

# Ponteiros (com passagens por "referência")

```
int ponteirosStart()
{
    printf("Hello ponteiros!\n");
    int i;
    i = 2;
    int *p;
    p = NULL;
    p = &i;
    funcao(p); //funcao(&i);
    printf("Endereco de p: %p\n", &p);
    printf("Endereco de i: %p\n", p);
    printf("Valor de i %d\n", *p);
    //printf("%p\n", &ponteiro);
    return 0;
}

void funcao(int *j){
    *j = 7;
}
```

|   | 1     | 2      | 3      | 4 | 5 | 6 |         |
|---|-------|--------|--------|---|---|---|---------|
| A | i (7) | p (A1) | j (A1) |   |   |   | *p = 7  |
| B |       |        |        |   |   |   | &p = A2 |
| C |       |        |        |   |   |   | &i = A1 |
| D |       |        |        |   |   |   | p = A1  |
| E |       |        |        |   |   |   | j = A1  |
| F |       |        |        |   |   |   | *j = 7  |

# Ponteiros (com passagens por "referência")

```
int ponteirosStart()
{
    printf("Hello ponteiros!\n");
    int i;
    i = 2;
    int *p;
    p = NULL;
    p = &i;
    funcao(p); //funcao(&i);
    printf("Endereco de p: %p\n", &p);
    printf("Endereco de i: %p\n", p);
    printf("Valor de i %d\n", *p);
    //printf("%p\n", &pinteiro);
    return 0;
}

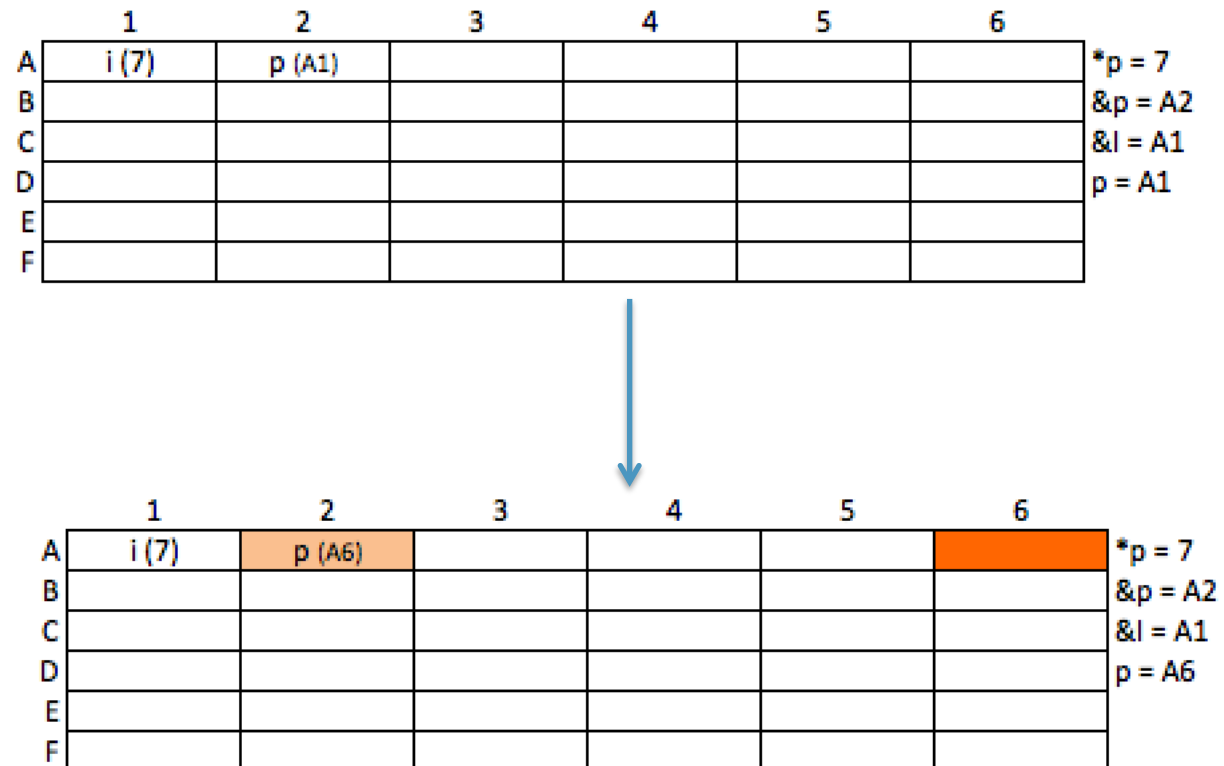
void funcao(int *j){
    *j = 7;
}
```

|   | 1     | 2      | 3 | 4 | 5 | 6 |         |
|---|-------|--------|---|---|---|---|---------|
| A | i (7) | p (A1) |   |   |   |   | *p = 7  |
| B |       |        |   |   |   |   | &p = A2 |
| C |       |        |   |   |   |   | &i = A1 |
| D |       |        |   |   |   |   | p = A1  |
| E |       |        |   |   |   |   |         |
| F |       |        |   |   |   |   |         |

# Ponteiros (alocação dentro de uma função)



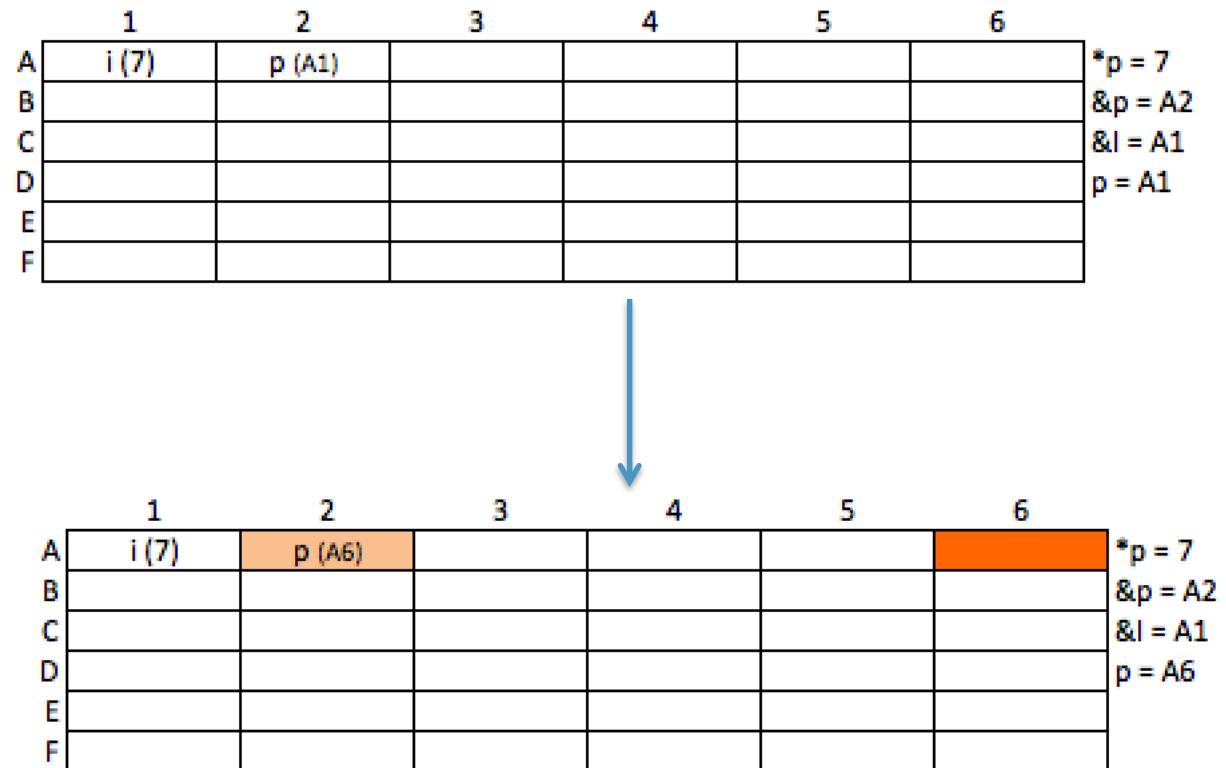
- Vai ser alocado um novo espaço de memória.
- O ponteiro p deve apontar para esse novo espaço
- Suponha que esse novo espaço seja alocado em A6
- Como fazer para p apontar para A6?



# Ponteiros (alocação dentro de uma função)



- Dentro da mesma função seria
  - `p = malloc(sizeof(int));`
- Como vai ser dentro de outra função, você quer mudar o conteúdo do ponteiro `p`, e não o conteúdo de onde `p` aponta
- Se para mudar o conteúdo de uma **variável normal** é preciso **um ponteiro (\*)**, então para mudar por “referência” o conteúdo de um **ponteiro** é preciso **um ponteiro para ponteiro (\*\*)**



# Ponteiros (alocação dentro de uma função)

```
int ponteirosStart()
{
    printf("Hello ponteiros!\n");
    int i;
    i = 2;
    int *p;
    p = NULL;
    p = &i;
    funcao(p, &p); // funcao(&i);
    printf("Endereco de p: %p\n", &p);
    printf("Endereco de i: %p\n", p);
    printf("Valor de i %d\n", *p);
    // printf("%p\n", &pinteiro);
    return 0;
}

void funcao(int *j, int **novo){
    *j = 7;
    int *t = NULL;
    t = malloc(sizeof(int));
    *t = 10;
    *novo = t;
}
```

|   | 1     | 2      | 3 | 4 | 5 | 6 |         |
|---|-------|--------|---|---|---|---|---------|
| A | i (7) | p (A1) |   |   |   |   | *p = 7  |
| B |       |        |   |   |   |   | &p = A2 |
| C |       |        |   |   |   |   | &i = A1 |
| D |       |        |   |   |   |   | p = A1  |
| E |       |        |   |   |   |   |         |
| F |       |        |   |   |   |   |         |

# Ponteiros (alocação dentro de uma função)



```
int ponteirosStart()
{
    printf("Hello ponteiros!\n");
    int i;
    i = 2;
    int *p;
    p = NULL;
    p = &i;
    funcao(p, &p); // funcao(&i);
    printf("Endereco de p: %p\n", &p);
    printf("Endereco de i: %p\n", p);
    printf("Valor de i %d\n", *p);
    // printf("%p\n", &ponteiro);
    return 0;
}

void funcao(int *j, int **novo){
    *j = 7;
    int *t = NULL;
    t = malloc(sizeof(int));
    *t = 10;
    *novo = t;
}
```

|   | 1     | 2      | 3      | 4        | 5 | 6 |         |
|---|-------|--------|--------|----------|---|---|---------|
| A | i (2) | p (A1) | j (A1) | novo(A2) |   |   | *p = 2  |
| B |       |        |        |          |   |   | &p = A2 |
| C |       |        |        |          |   |   | &i = A1 |
| D |       |        |        |          |   |   | p = A1  |
| E |       |        |        |          |   |   | j = A1  |
| F |       |        |        |          |   |   | *j = 2  |

novo = A2  
 \*novo = A1  
 \*\*novo = 2

# Ponteiros (alocação dentro de uma função)



```
int ponteirosStart()
{
    printf("Hello ponteiros!\n");
    int i;
    i = 2;
    int *p;
    p = NULL;
    p = &i;
    funcao(p, &p); //funcao(&i);
    printf("Endereco de p: %p\n", &p);
    printf("Endereco de i: %p\n", p);
    printf("Valor de i %d\n", *p);
    //printf("%p\n", &ponteiro);
    return 0;
}

void funcao(int *j, int **novo){
    *j = 7;
    int *t = NULL;
    t = malloc(sizeof(int));
    *t = 10;
    *novo = t;
}
```

|   | 1     | 2      | 3      | 4        | 5 | 6 |         |
|---|-------|--------|--------|----------|---|---|---------|
| A | i (7) | p (A1) | j (A1) | novo(A2) |   |   | *p = 7  |
| B |       |        |        |          |   |   | &p = A2 |
| C |       |        |        |          |   |   | &i = A1 |
| D |       |        |        |          |   |   | p = A1  |
| E |       |        |        |          |   |   | j = A1  |
| F |       |        |        |          |   |   | *j = 7  |

novo = A2  
 \*novo = A1  
 \*\*novo = 7

# Ponteiros (alocação dentro de uma função)



```

int ponteirosStart()
{
    printf("Hello ponteiros!\n");
    int i;
    i = 2;
    int *p;
    p = NULL;
    p = &i;
    funcao(p, &p); // funcao(&i);
    printf("Endereco de p: %p\n", &p);
    printf("Endereco de i: %p\n", p);
    printf("Valor de i %d\n", *p);
    // printf("%p\n", &ponteiro);
    return 0;
}

void funcao(int *j, int **novo){
    *j = 7;
    int *t = NULL;
    t = malloc(sizeof(int));
    *t = 10;
    *novo = t;
}

```

|   | 1     | 2      | 3      | 4        | 5          | 6 |         |            |
|---|-------|--------|--------|----------|------------|---|---------|------------|
| A | i (7) | p (A1) | j (A1) | novo(A2) | t(NULL(0)) |   | *p = 7  | novo = A2  |
| B |       |        |        |          |            |   | &p = A2 | *novo = A1 |
| C |       |        |        |          |            |   | &i = A1 | **novo = 7 |
| D |       |        |        |          |            |   | p = A1  | t = NULL   |
| E |       |        |        |          |            |   | j = A1  | *t = 0     |
| F |       |        |        |          |            |   | *j = 7  |            |

# Ponteiros (alocação dentro de uma função)

```
int ponteirosStart()
{
    printf("Hello ponteiros!\n");
    int i;
    i = 2;
    int *p;
    p = NULL;
    p = &i;
    funcao(p, &p); //funcao(&i);
    printf("Endereco de p: %p\n", &p);
    printf("Endereco de i: %p\n", p);
    printf("Valor de i %d\n", *p);
    //printf("%p\n", &pinteiro);
    return 0;
}

void funcao(int *j, int **novo){
    *j = 7;
    int *t = NULL;
    t = malloc(sizeof(int));
    *t = 10;
    *novo = t;
}
```

@@@ = lixo. O conteúdo de A6 é lixo.

|   | 1     | 2      | 3      | 4        | 5     | 6   |         |            |
|---|-------|--------|--------|----------|-------|-----|---------|------------|
| A | i (7) | p (A1) | j (A1) | novo(A2) | t(A6) | @@@ | *p = 7  | novo = A2  |
| B |       |        |        |          |       |     | &p = A2 | *novo = A1 |
| C |       |        |        |          |       |     | &i = A1 | **novo = 7 |
| D |       |        |        |          |       |     | p = A1  | t = A6     |
| E |       |        |        |          |       |     | j = A1  | *t = @@@   |
| F |       |        |        |          |       |     | *j = 7  |            |

# Ponteiros (alocação dentro de uma função)



```
int ponteirosStart()
{
    printf("Hello ponteiros!\n");
    int i;
    i = 2;
    int *p;
    p = NULL;
    p = &i;
    funcao(p, &p); //funcao(&i);
    printf("Endereco de p: %p\n", &p);
    printf("Endereco de i: %p\n", p);
    printf("Valor de i %d\n", *p);
    //printf("%p\n", &pinteiro);
    return 0;
}

void funcao(int *j, int **novo){
    *j = 7;
    int *t = NULL;
    t = malloc(sizeof(int));
    *t = 10;
    *novo = t;
}
```

|   | 1     | 2      | 3      | 4        | 5     | 6  |         |            |
|---|-------|--------|--------|----------|-------|----|---------|------------|
| A | i (7) | p (A1) | j (A1) | novo(A2) | t(A6) | 10 | *p = 7  | novo = A2  |
| B |       |        |        |          |       |    | &p = A2 | *novo = A1 |
| C |       |        |        |          |       |    | &i = A1 | **novo = 7 |
| D |       |        |        |          |       |    | p = A1  | t = A6     |
| E |       |        |        |          |       |    | j = A1  | *t = 10    |
| F |       |        |        |          |       |    | *j = 7  |            |

# Ponteiros (alocação dentro de uma função)



```
int ponteirosStart()
{
    printf("Hello ponteiros!\n");
    int i;
    i = 2;
    int *p;
    p = NULL;
    p = &i;
    funcao(p, &p); // funcao(&i);
    printf("Endereco de p: %p\n", &p);
    printf("Endereco de i: %p\n", p);
    printf("Valor de i %d\n", *p);
    // printf("%p\n", &ponteiro);
    return 0;
}

void funcao(int *j, int **novo){
    *j = 7;
    int *t = NULL;
    t = malloc(sizeof(int));
    *t = 10;
    *novo = t;
}
```

|   | 1     | 2      | 3      | 4        | 5     | 6  |         |            |
|---|-------|--------|--------|----------|-------|----|---------|------------|
| A | i (7) | p (A6) | j (A1) | novo(A2) | t(A6) | 10 | *p = 10 | novo = A2  |
| B |       |        |        |          |       |    | &p = A2 | *novo = A6 |
| C |       |        |        |          |       |    | &i = A1 | **novo = 7 |
| D |       |        |        |          |       |    | p = A6  | t = A6     |
| E |       |        |        |          |       |    | j = A1  | *t = 10    |
| F |       |        |        |          |       |    | *j = 7  |            |

# Ponteiros (alocação dentro de uma função)

```
int ponteirosStart()
{
    printf("Hello ponteiros!\n");
    int i;
    i = 2;
    int *p;
    p = NULL;
    p = &i;
    funcao(p, &p); // funcao(&i);
    printf("Endereco de p: %p\n", &p);
    printf("Endereco de i: %p\n", p);
    printf("Valor de i %d\n", *p);
    // printf("%p\n", &pinteiro);
    return 0;
}

void funcao(int *j, int **novo){
    *j = 7;
    int *t = NULL;
    t = malloc(sizeof(int));
    *t = 10;
    *novo = t;
}
```

|   | 1     | 2      | 3 | 4 | 5 | 6  |         |
|---|-------|--------|---|---|---|----|---------|
| A | i (7) | p (A6) |   |   |   | 10 | *p = 10 |
| B |       |        |   |   |   |    | &p = A2 |
| C |       |        |   |   |   |    | &i = A1 |
| D |       |        |   |   |   |    | p = A6  |
| E |       |        |   |   |   |    |         |
| F |       |        |   |   |   |    |         |

# Referências



- Material compilado de:
  - C completo e total
  - Ponteiros, Monitoria de introdução a programação, Cin UFPE.  
[www.cin.ufpe.br/~if669ec/files/AP8.pptx](http://www.cin.ufpe.br/~if669ec/files/AP8.pptx).
  - <http://www.ime.usp.br/~pf/algoritmos/aulas/pont.html>