


Centro Universitário de Araraquara – UNIARA

Departamento de Ciências da Administração e Tecnologia

Curso de Engenharia de Computação e Sistemas de Informação



Estrutura de Dados Ponteiros



Prof. Msc. José Eduardo Ribeiro

Ponteiros

- ▶ A linguagem C permite o **armazenamento** e a **manipulação de valores de endereços de memória**
- ▶ Para cada **tipo de dado** existente, há um **tipo ponteiro** que pode armazenar endereços de memória onde existem valores do tipo correspondente armazenados
 - ▶ Por exemplo:
 - ▶ Se **x** for declarado como um ponteiro, **&x** se referirá à posição de memória reservada para **conter** um **endereço de memória** do mesmo **tipo**



Ponteiros

- ▶ Tipos primitivos:

```
int a;
```

- ▶ Na declaração acima, teríamos um endereçamento similar a:

```
0x00000000 | -> Início da memória relativa à variável "a"  
0x00000002  
0x00000003  
0x00000004 -> Fim da memória relativa à variável "a"
```



Variáveis Ponteiros

- ▶ Declaração:

```
tipo *variavel;
```

- ▶ Exemplos

```
char    *ch;  
int     *i;  
float   *num;
```



Ponteiros

- ▶ **Quando escrevemos:**

- ▶ `int a;`

- ▶ Declaramos uma variável com nome **a** que pode armazenar valores inteiros

- ▶ Automaticamente, reserva-se um espaço na memória suficiente para armazenar valores inteiros (geralmente 4 bytes)

- ▶ Da mesma forma que declaramos **variáveis** para **armazenar inteiros**, podemos declarar variáveis que, em vez de servirem para armazenar valores inteiros, servem para armazenar **valores de endereços de memória** onde há **variáveis inteiras referenciadas**



Ponteiros

▶ Exemplo

- ▶ `int *p;`
- ▶ Declaramos uma variável ponteiro do tipo inteiro com nome **p** que pode armazenar endereços de memória onde existe um inteiro alocado
- ▶ C não reserva uma palavra especial para a declaração de ponteiros
 - ▶ Usamos a mesma palavra do tipo com os nomes das variáveis precedidas pelo caractere *
 - ▶ Neste caso, declaramos uma variável com nome **p** que pode armazenar endereços de memória onde existe um inteiro armazenado



Ponteiros

- ▶ Existe um operador, muito útil, que permite saber o tamanho, em bytes, de um tipo de dado, variável ou estrutura: **sizeof()**

```
#include <stdio.h>
void main() {
    printf("%d\n",sizeof(char));
    printf("%d\n",sizeof(int));
    printf("%d\n",sizeof(double));
    system("PAUSE");
}
```

Saída do programa:

```
1
4
8
```



Ponteiros

- ▶ O tamanho de um ponteiro é sempre 4 bytes (32 bits)

```
#include <stdio.h>
void main() {
    printf("%d\n",sizeof(char *));
    printf("%d\n",sizeof(int *));
    printf("%d\n",sizeof(double *));
    system("PAUSE");
}
```

Saída do programa:

```
4
4
4
```


Ponteiros

- ▶ C oferece dois operadores unários para atribuir e acessar endereços de memória
 - ▶ O operador unário **&** (“endereço de”), aplicado a variáveis, resulta no endereço da posição da memória reservada para a variável
 - ▶ O operador unário ***** (“conteúdo de”), aplicado as variáveis do tipo ponteiro, acessa o conteúdo do endereço de memória armazenado pela variável ponteiro



Ponteiros

- ▶ O operador & retorna o endereço de memória de uma variável, estrutura ou função que o sucede

```
#include <stdio.h>
int main() {
    int exemplo;
    // O formatador %p é utilizado para mostrar endereços de memória
    printf("%p",&exemplo);
    system("PAUSE");
return 0;
}
```



Ponteiros

► Exemplos;

```
#include <stdio.h>
int main() {
    int exemplo; // Uma variável do tipo inteiro
    int * ponteiro; // Um ponteiro para uma variável do tipo inteiro
    printf("Endereco de 'exemplo' : %p\n", &exemplo);
    printf("Endereco de 'ponteiro': %p\n", &ponteiro);
    system("PAUSE");
    return 0;
}
```



Variáveis Ponteiros - Notação

► Notação

- `int a;` *//variável inteiro*
- `int *p;` *//variável ponteiro para inteiro*

► `int *p;`

- a declaração `p = &a`
 - o ponteiro **p** recebe o endereço da variável **a**
- a notação `*p` em C
 - refere-se ao *inteiro* na posição referenciada pelo ponteiro **p**



Variáveis Ponteiros - Notação

▶ Exemplos

- ▶ `int a;` `// variável inteira`
- ▶ `int *p;` `//variável ponteiro do tipo inteira`

- ▶ `a = 5 ;` `//a recebe o valor 5`
- ▶ `p = &a;` `// p recebe o endereço de a (diz-se p aponta para a)`

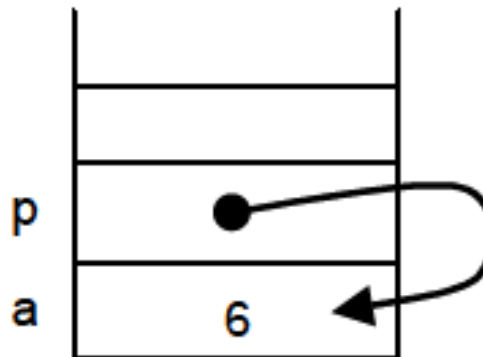
- ▶ `*p = 6;` `//conteúdo de p recebe o valor 6`

- ▶ Acessar **a** é equivalente a acessar ***p**, pois **p** armazena o endereço de **a**. Dizemos que **p** aponta para **a**, daí o nome ponteiro



Ponteiros


- ▶ Em vez de criarmos valores fictícios para os endereços de memória no nosso esquema ilustrativo da memória, podemos desenhar setas graficamente, sinalizando que um ponteiro aponta para uma determinada variável
- ▶ Exemplos anterior
 - ▶ `*p = 6;` //conteúdo apontado por p recebe o valor 6



O que é então Ponteiro?

- ▶ É uma variável que contém um endereço de memória do mesmo tipo referenciado
- ▶ Usualmente esse é o endereço de outra variável na memória

Endereço de memória	Valor variável na memória
⋮	⋮
112	Eduardo
108	
104	
100	112



Para facilitar e não se preocupar com valores hexadecimais, vamos considerar valores inteiros ao se referenciar área de memória.

Operador Unário &

- Devolve o endereço de memória de seu operando

```
int a;  
int *pa;  
  
a = 5;  
  
pa = &a;
```

Variável	Endereço	Memória
pa	1008	?
a	1000	5
		⋮

Operador Unário &

- Devolve o endereço de memória de seu operando

```
int a;  
int *pa;  
  
a = 5;  
  
pa = &a;
```

Variável	Endereço	Memória
----------	----------	---------

pa	1008	1000
a	1000	5
		⋮



Operador Unário *

- Devolve o valor da variável localizada no endereço de seu operando

```
int a, b;
```

```
int *pa;
```

```
a = 5;
```

```
pa = &a;
```

```
b = *pa;
```

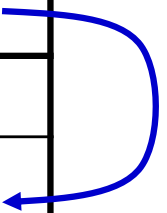
Variável	Endereço	Memória
----------	----------	---------

b	1012	?
---	------	---

pa	1008	1000
----	------	------

a	1000	5
---	------	---

		⋮
--	--	---



Operador Unário *

- Devolve o valor da variável localizada no endereço de seu operando

```
int a, b;
```

```
int *pa;
```

```
a = 5;
```

```
pa = &a;
```

```
b = *pa;
```

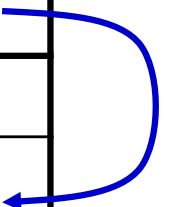
Variável	Endereço	Memória
----------	----------	---------

b	1012	5
---	------	---

pa	1008	1000
----	------	------

a	1000	5
---	------	---

		⋮
--	--	---



Operador Unário *

- Devolve o valor da variável localizada no endereço de seu operando

```
int a, b;  
int *pa;
```

```
a = 5;  
pa = &a;  
b = *pa;
```

```
*pa = 10;
```

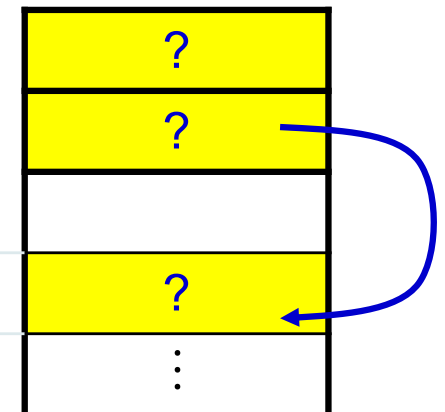
Variável	Endereço	Memória
----------	----------	---------

b	1012	?
---	------	---

pa	1008	?
----	------	---

a	1000	?
---	------	---

		⋮
--	--	---



Operador Unário *

- Devolve o valor da variável localizada no endereço de seu operando

```
int a, b;  
int *pa;
```

```
a = 5;  
pa = &a;  
b = *pa;
```

```
*pa = 10;
```

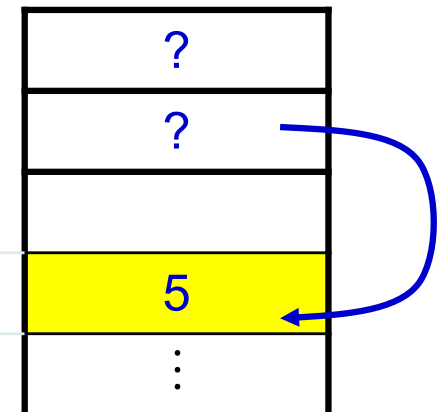
Variável	Endereço	Memória
----------	----------	---------

b	1012	?
---	------	---

pa	1008	?
----	------	---

a	1000	5
---	------	---

		⋮
--	--	---



Operador Unário *

- ▶ Devolve o valor da variável localizada no endereço de seu operando

```
int a, b;  
int *pa;
```

```
a = 5;  
pa = &a;  
b = *pa;
```

```
*pa = 10;
```

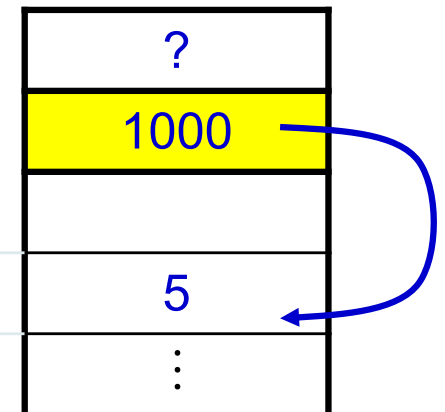
Variável	Endereço	Memória
----------	----------	---------

b	1012	?
---	------	---

pa	1008	1000
----	------	------

a	1000	5
---	------	---

		⋮
--	--	---



Operador Unário *

- Devolve o valor da variável localizada no endereço de seu operando

```
int a, b;  
int *pa;
```

```
a = 5;  
pa = &a;  
b = *pa;
```

```
*pa = 10;
```

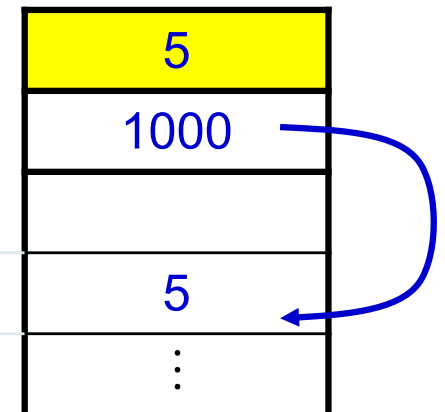
Variável	Endereço	Memória
----------	----------	---------

b	1012	5
---	------	---

pa	1008	1000
----	------	------

a	1000	5
---	------	---

		⋮
--	--	---



Operador Unário *

- ▶ Devolve o valor da variável localizada no endereço de seu operando

```
int a, b;  
int *pa;
```

```
a = 5;  
pa = &a;  
b = *pa;
```

Variável	Endereço	Memória
----------	----------	---------

b	1012	5
---	------	---

pa	1008	1000
----	------	------

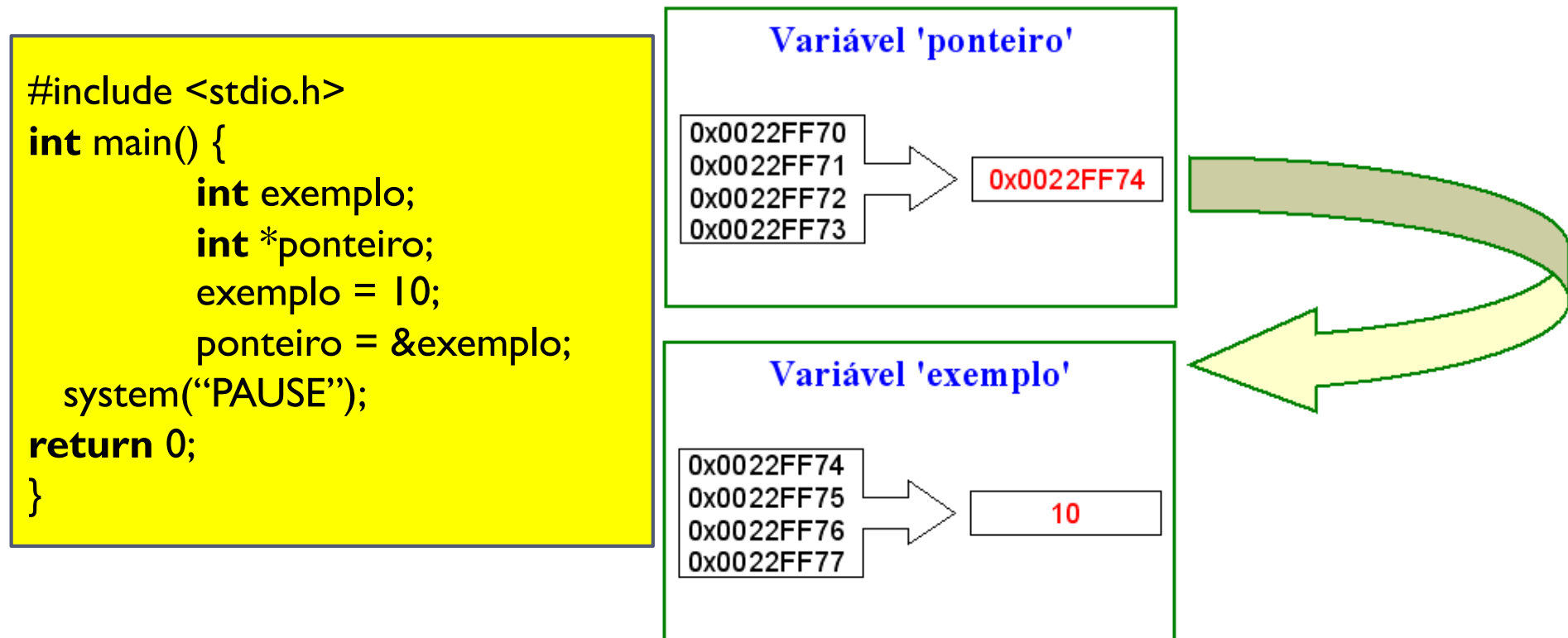
a	1000	10
---	------	----

⋮

```
*pa = 10;
```


Ponteiros

- ▶ O tamanho de um ponteiro é sempre 4 bytes (32 bits) e armazena o início de memória da variável referenciada



Exemplo

- ▶ Programa que altera o valor de uma variável **a** inteiro com um ponteiro

```
# include<stdio.h>

void main(void){

    int a;
    int *p;

    p = &a;
    *p = 2;

    printf(" %d ", a);

}
```

Exemplo

- ▶ Exemplo abaixo mostra a operação errada com ponteiros (aponta para uma variável que não está no contexto) ou (lixo de memória)

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

```
void main(void){
```

```
    int a, b, *p;
```

```
    a = 2;
```

```
    *p = 3;
```

```
    b = a + (*p);
```

```
    printf(" %d ", b);
```

```
}
```

Deseja-se atribuir o valor 3 a variável b com auxílio de um ponteiro.

Exemplo

- ▶ Exemplo abaixo mostra a operação errada com ponteiros (aponta para uma variável que não está no contexto) ou (lixo de memória)

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

```
void main(void){
```

```
    int a, b, *p;
```

```
    a = 2;
```

```
    *p = 3;
```

```
    b = a + (*p);
```

```
    printf(" %d ", b);
```

```
}
```

Erro!

Só podemos alterar o conteúdo apontado por um ponteiro se este tiver sido devidamente inicializado

Ele deve apontar para um espaço de memória onde já se prevê o armazenamento de valores do tipo em questão

Exemplo

- ▶ Corrigindo o código para alterar o valor da variável **a**

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

```
void main(void){
```

```
    int a, b, *p;
```

```
    a = 2;
```

```
    p =&b;
```

```
    *p = 3;
```

```
    b = a + (*p);
```

```
    printf(" %d ", b);
```

```
}
```

Correto!

Só podemos alterar o conteúdo apontado por um ponteiro se este tiver sido devidamente inicializado

Ele deve apontar para um espaço de memória onde já se prevê o armazenamento de valores do tipo em questão

Valor de x ?

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

```
int main(void) {
```

```
    int x;
    int *i;
```

```
    x = 23;
    *i = 19;
```

```
    printf("x = %d", x);
    printf("i = %d", *i);
```

```
    system("PAUSE");
    return 0;
```

```
}
```

```
Projeto(ex_ponteiro_new)
```

Valor de i ?

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

```
int main(void) {
```

```
    int x;
    int *i;
```

```
    x = 23;
    i = &x;
    *i = 19;
    printf("x = %d", x);
    printf("i = %d", *i);
```

```
    system("PAUSE");
    return 0;
```

```
}
```

```
Projeto(ex_ponteiro_new2)
```

Diga qual é a saída dos programas?

Valor de i ?

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

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

```
int main(void) {
```

```
    int x;
```

```
    int *i;
```

```
    x = 23;
```

```
    *i = 19;
```



**Ponteiro
Selvagem**

```
    printf("x = %d", x);
```

```
    printf("i = %d", *i);
```

```
    system("PAUSE");
```

```
    return 0;
```

```
}
```



Valor de i ?

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

int main(void) {

    int x;
    int *i;

    x = 23;
    i = &x;
    *i = 19;

    printf("x = %d", x);
    printf("\ni = %d", *i);

    system("PAUSE");
    return 0;
}
```

Solução !

O ponteiro aponta agora para a variável **x**, portanto podemos manipular o que esta armazenado em **x** pelo ponteiro **i**.

Aritmética de ponteiros

► Adição e subtração

```
char *ch = 1000;  
int *i = 1000;
```

Para:

char: 1 byte

int: 4 bytes

Tipo char	Endereço Memória	Tipo int
ch	1000	i
ch + 1	1001	
ch + 2	1002	
ch + 3	1003	
ch + 4	1004	i + 1
ch + 5	1005	
ch + 6	1006	
ch + 7	1007	
	⋮	

Relativo ao tipo primitivo

Aritmética de ponteiros

► Adição e subtração

```
char *ch = 1000;  
int *i = 1000;
```

Tipo char	Endereço Memória	Tipo int
ch	1000	i
ch + 1	1001	
ch + 2	1002	
ch + 3	1003	
ch + 4	1004	i + 1
ch + 5	1005	
ch + 6	1006	
ch + 7	1007	
	⋮	

Para:

char: 1 byte

int: 4 bytes

Relativo ao seu tipo base

Ponteiros

► Lógica de endereçamento

1. Se **pi** é um ponteiro para um tipo inteiro, então **pi+1** é o ponteiro (endereço de memória) para o inteiro imediatamente seguinte ao inteiro ***pi** em memória
 - Por exemplo, se o endereço de **pi** é 100 (isto é, **pi** aponta para o inteiro ***pi** na posição 100) então **pi+1** é 104
2. ***pi+1** se refere a 1 somado ao inteiro ***pi**
3. ***(pi+1)** se refere ao inteiro posterior ao inteiro na posição **pi**
 - Por exemplo: **pi** tem posição 100, **(pi+1)** tem a posição 104, ***(pi+1)** é o valor inteiro que está na posição (104)



Ponteiros

▶ Exemplo

- ▶ `include <stdio.h>`
- ▶ `void imprime_array_elemento (int *paArray, int piElemento) {`
- ▶ `int elemento;`
- ▶ `elemento = *(paArray + piElemento-1);`
- ▶ `printf("Elemento %d = %d\n", piElemento, elemento);`
- ▶ `}`

- ▶ `void main(void)`
- ▶ `{`
- ▶ `int aArray[] = { 10, 20, 30, 40, 50, 60 };`
- ▶ `imprime_array_elemento(aArray,1);`
- ▶ `imprime_array_elemento(aArray,3);`
- ▶ `system("PAUSE");`
- ▶ `}`



Funções

- Chamada por valor
- Chamada por referência



Ponteiros (Passagem de Parâmetros)

▶ Passagem de parâmetro por valor

- ▶ Quando o valor de um parâmetro for alterado dentro da função, o valor no programa de chamada não será modificado

```
...  
▶ x=5;  
▶ printf(“%d\n”, x);  
▶ funct(x);  
▶ printf (“%d\n”, x);  
▶ ...  
▶ funct(int y) {  
▶     ++y;  
▶     printf(“%d\n”, y);  
▶ }  
▶ ...
```

Já descrevemos que as **funções** não podem **alterar** diretamente **valores** de **variáveis** da função que fez a chamada

Exceto com o uso de arrays
(Referência Simulada)



Chamada por valor

► Exemplo

```
#include <stdio.h>

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

int main(void) {
    int val1, val2, resp;
    val1= 10;
    val2= 15;
    resp = soma(val1, val2);
    printf("soma= %d", resp);
    system("PAUSE");
    return 0;
}
```



Chamada por valor

► Exemplo

```
#include <stdio.h>
#define EUA 1
#define BRA 0

void MostraData(int dia, int mes, int ano, char modo){
    if (modo == EUA)
        printf("%d/%d/%d",mes, dia, ano);
    else
        printf("%d/%d/%d",dia, mes, ano);
}

int main(void){
    int d,m,a;
    d= 22;
    m= 12;
    a= 1968;
    MostraData(d, m, a, EUA);
    system("PAUSE");
    return 0;
}
```



Ponteiros (Passagem de Parâmetros)

- ▶ Passagem de parâmetro por referência
 - ▶ Os ponteiros representam o mecanismo usado em C para permitir que uma função chamada **modifique** valores de variáveis numa função de chamada (ou chamadora)

- ▶ ...
- ▶ x=5;
- ▶ printf(“%d\n”, x);
- ▶ funct(&x);
- ▶ printf (“%d\n”, x);
- ▶ ...
- ▶ funct(int *y) {
- ▶ *y = 6;
- ▶ printf(“%d”, *y);
- ▶ }
- ▶ ...

Se passarmos para uma função o **valor do endereço de memória** onde a variável está alocada, a função pode **alterar** indiretamente o valor da variável da função que a chamou



Chamada por referência

► Exemplo

```
#include <stdio.h>

void troca(int *a, int *b) {
    int temp;
    temp= *a;
    *a= *b;
    *b= temp;
}

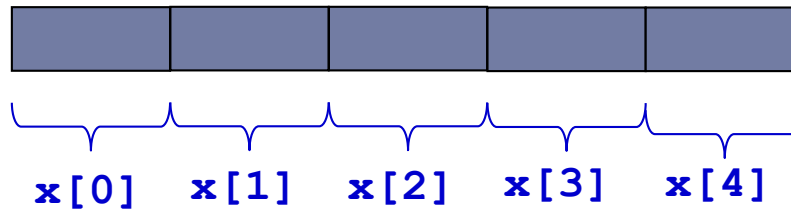
int main(void) {
    int val1, val2;
    val1= 10;
    val2= 15;
    printf("val1= %d    val2= %d \n", val1, val2);
    troca(&val1,&val2);
    printf("val1= %d    val2= %d \n", val1, val2);
    system("PAUSE");
    return 0;
}
```



Revisão

Vetores em C

`int x[5];` Posições contíguas de memória

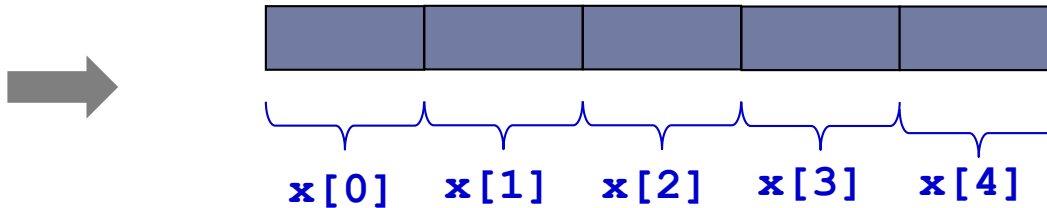


Variável	Endereço	Memória
x[4]	1016	
x[3]	1012	
x[2]	1008	
x[1]	1004	
x[0]	1000	
		⋮

Revisão

Vetores em C

`int x[5];` Posições contíguas de memória



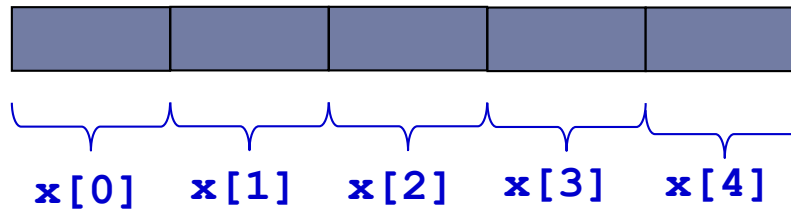
```
int i;  
for (i= 0; i < 5; i++){  
    x[i]= i*10;  
}
```

Variável	Endereço	Memória
<code>x[4]</code>	1016	
<code>x[3]</code>	1012	
<code>x[2]</code>	1008	
<code>x[1]</code>	1004	
<code>x[0]</code>	1000	
		⋮

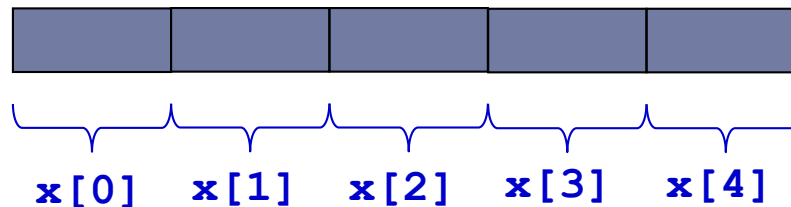
Revisão

Vetores em C

`int x[5];` Posições contíguas de memória



```
int i;  
for (i= 0; i < 5; i++){  
    x[i]= i*10;  
}
```



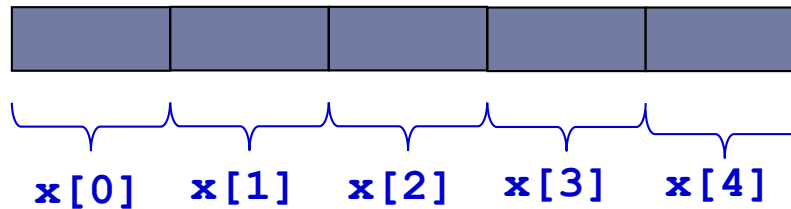
Variável	Endereço	Memória
<code>x[4]</code>	1016	
<code>x[3]</code>	1012	
<code>x[2]</code>	1008	
<code>x[1]</code>	1004	
<code>x[0]</code>	1000	
		⋮



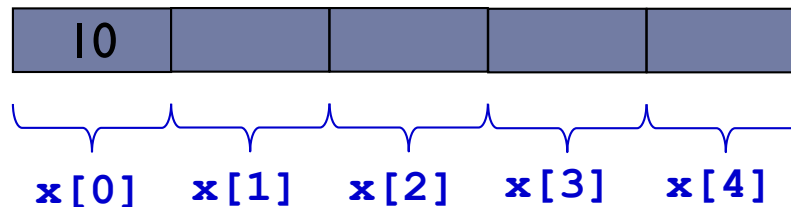
Revisão

Vetores em C

`int x[5];` Posições contíguas de memória



```
int i;  
for (i= 0; i < 5; i++){  
    x[i]= i*10;  
}
```



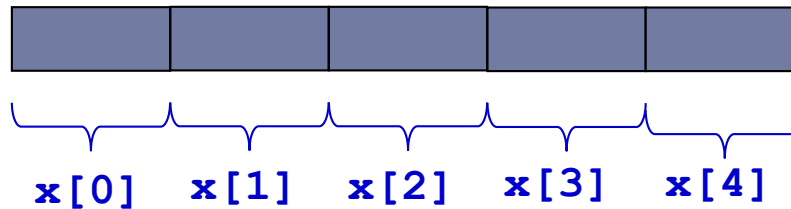
Variável	Endereço	Memória
<code>x[4]</code>	1016	
<code>x[3]</code>	1012	
<code>x[2]</code>	1008	
<code>x[1]</code>	1004	
<code>x[0]</code>	1000	10
		⋮



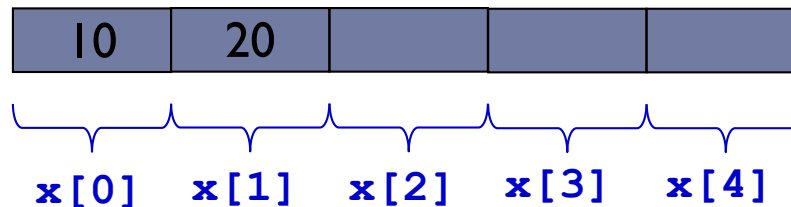
Revisão

Vetores em C

`int x[5];` Posições contíguas de memória



```
int i;  
for (i= 0; i < 5; i++){  
    x[i]= i*10;  
}
```



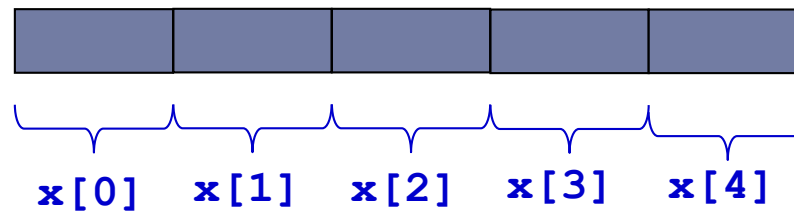
Variável	Endereço	Memória
<code>x[4]</code>	1016	
<code>x[3]</code>	1012	
<code>x[2]</code>	1008	
<code>x[1]</code>	1004	20
<code>x[0]</code>	1000	10
		⋮



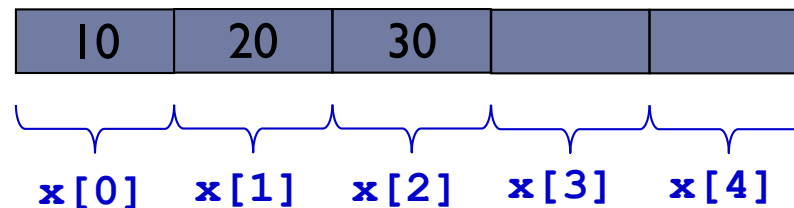
Revisão

Vetores em C

`int x[5];` Posições contíguas de memória



```
int i;  
for (i= 0; i < 5; i++){  
    x[i]= i*10;  
}
```



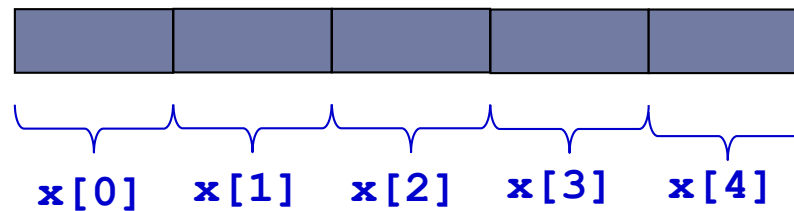
Variável	Endereço	Memória
<code>x[4]</code>	1016	
<code>x[3]</code>	1012	
<code>x[2]</code>	1008	30
<code>x[1]</code>	1004	20
<code>x[0]</code>	1000	10
		⋮



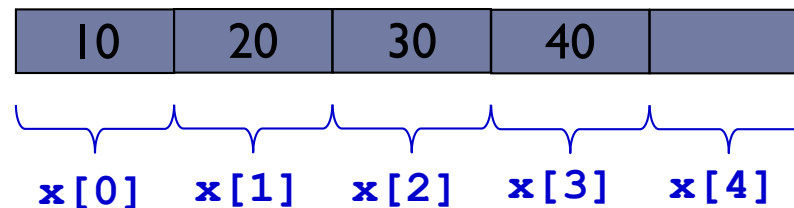
Revisão

Vetores em C

`int x[5];` Posições contíguas de memória



```
int i;  
for (i= 0; i < 5; i++){  
    x[i]= i*10;  
}
```

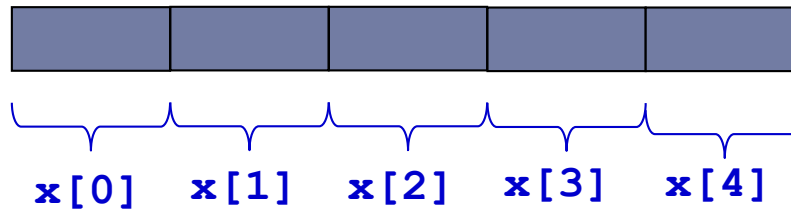


Variável	Endereço	Memória
<code>x[4]</code>	1016	
<code>x[3]</code>	1012	40
<code>x[2]</code>	1008	30
<code>x[1]</code>	1004	20
<code>x[0]</code>	1000	10
		⋮

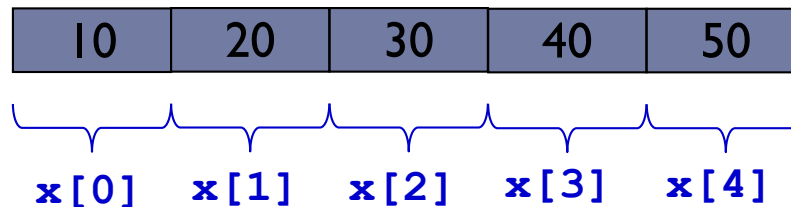
Revisão

Vetores em C

`int x[5];` Posições contíguas de memória



```
int i;  
for (i= 0; i < 5; i++){  
    x[i] = i*10;  
}
```

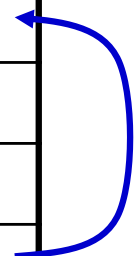


Variável	Endereço	Memória
<code>x[4]</code>	1016	50
<code>x[3]</code>	1012	40
<code>x[2]</code>	1008	30
<code>x[1]</code>	1004	20
<code>x[0]</code>	1000	10
		⋮

Vetor é um ponteiro em C

```
int x[5];  
int *p;  
  
p = x;  
// p aponta para a primeira  
posição de x  
  
x[0]= 15; // *p é igual a 15;
```

Variável	Endereço	Memória
x[4]	1016	40
x[3]	1012	30
x[2]	1008	20
x[1]	1004	10
x[0]	1000	1
		:
p	900	1000



Exemplo 1

```
#include <stdio.h>
int main(void) {
```

```
    int x[5]={10,15,12,17,14};
    int *p,*s;
```

```
    p = x;
    s = p + 1;
```

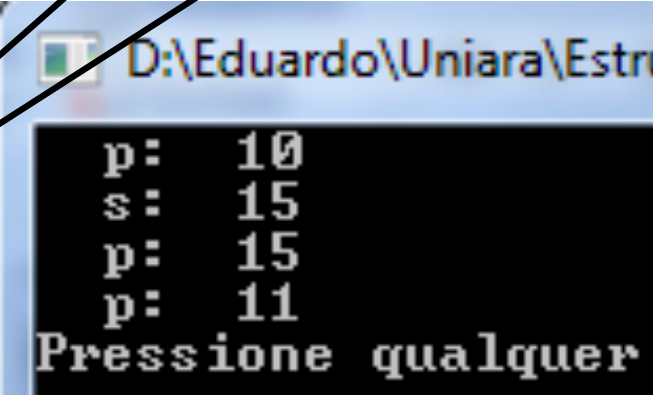
```
    printf("  p:  %d \n", *p) ;
    printf("  s:  %d \n", *s) ;
    printf("  p:  %d \n", *(p+1)) ;
    printf("  p:  %d \n", *p+1) ;
```

```
    system("PAUSE") ;
    return 0;
```

```
}
```

```
Projeto(exemplo_vetor_ponteiro)
```

Manipulações
diferenciadas com
ponteiros



```
D:\Eduardo\Uniara\Estru
p: 10
s: 15
p: 15
p: 11
Pressione qualquer
```

Exemplo2

```
int main(void) {  
  
    int x[5]={10,15,12,17,14};  
    int *p;  
    p = x;  
  
    printf(" p: %d \n", *p);  
    printf(" p: %d \n", *(++p));  
  
    getch();  
    return 0;  
}
```

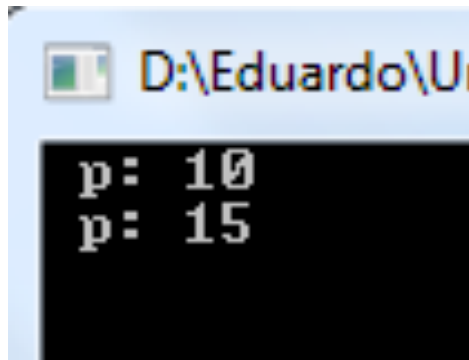
```
int main(void) {  
  
    int x[5]={10,15,12,17,14};  
    int *p;  
    p = x;  
  
    printf(" p: %d \n", *p);  
    printf(" p: %d \n", *(p++));  
  
    getch();  
    return 0;  
}
```

Qual a saída e a diferença dos exemplos acima?

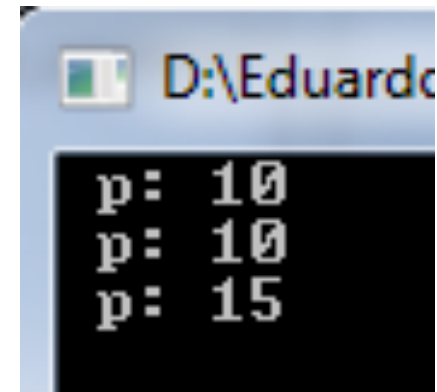


Exemplo2

```
int main(void){  
  
    int x[5]={10,15,12,17,14};  
    int *p;  
    p = x;  
  
    printf(" p: %d \n", *p);  
    printf(" p: %d \n", *(++p));  
  
    getch();  
    return 0;  
}
```



```
int main(void){  
  
    int x[5]={10,15,12,17,14};  
    int *p;  
    p = x;  
  
    printf(" p: %d \n", *p);  
    printf(" p: %d \n", *(p++));  
    printf(" p: %d \n", *p);  
  
    getch();  
    return 0;  
}
```



Respostas

Chamada por referência - vetores

```
#include <stdio.h>

void troca_AE(char str[]){
    int i;
    for(i= 0; i < strlen(str); i++)
        if (str[i] == 'A')
            str[i]= 'E';
}

int main(void){
    char ch[10]= "ccAAccAA";
    printf("%s \n", ch);
    troca_AE(ch);
    printf("%s", ch);
    getch();
    return 0;
}
```

Projeto(exemplo_vetor_ponteiro3)



Exercícios

- ▶ Crie um programa com 2 variáveis do tipo ponteiro (int *piValor e char *pcValor) e 1 variável do tipo inteiro (int iValor) e 1 variável do tipo char (cValor).
- ▶ Obtenha o valor de memória das variáveis iValor e cValor guardando nos respectivos ponteiros piValor e pcValor
- ▶ Imprima o endereço de memória armazenado nas variáveis ponteiro
- ▶ Incremente as variáveis ponteiro para obter a próxima posição de memória somados ao conteúdo armazenado
- ▶ Imprima novamente o endereço de memória armazenado nas variáveis ponteiro
 - ▶ Projeto(ponteiro3)



Exercícios

- ▶ Crie um programa que entre com 2 valores do tipo inteiro e imprima a soma destes números em uma variável resultado do tipo inteiro. Explore no programa os seguintes itens:
 - ▶ Imprima o endereço de memória da variável resultado
 - ▶ Crie uma função “soma(int piValorA, int piValorB, int *piResultado)” que some os valores de dois inteiros passados como valor, ou seja, as 2 primeiras variáveis e como referência a posição de memória da variável resultado. O ponteiro para resultado deve receber os valores somados de piValorA e piValorB.
 - ▶ Imprima o endereço de memória apontada pela variável ponteiro piResultado.
 - ▶ Imprima o endereço de memória da variável ponteiro piResultado
 - ▶ Imprima na tela o valor que aponta a variável do tipo ponteiro *piResultado
 - ▶ Projeto(ponteiro2)



Exercícios

- ▶ Inicialize um vetor de inteiros aleatoriamente e percorra e mostre os valores de todo o vetor usando dois ponteiros: um começando do início do vetor e outro do final
 - ▶ Projeto(Ex_ponteiro_ex6)



Exercícios

- ▶ Implemente uma função que calcule a área da superfície e o volume de uma esfera de raio r .
- ▶ E a função deve obedecer ao protótipo:
 - ▶ `void esfera (float r, float* area, float* volume);`
- ▶ A área da superfície e o volume são dados, respectivamente por $4\pi r^2$ e $4\pi r^3 / 3$.
 - ▶ Projeto_ex9

