



Universidade de Brasília (UnB)
Departamento de Ciência da Computação (CIC)

Ponteiros

116319 - Estruturas de Dados

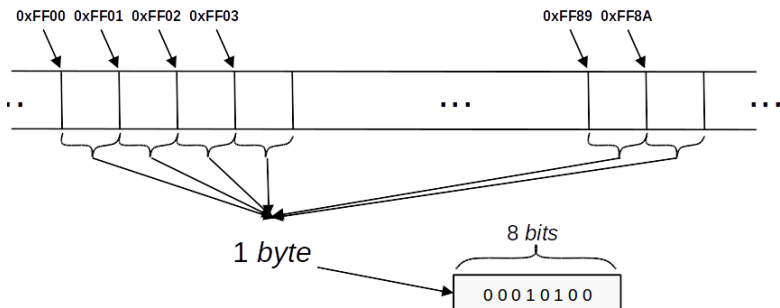
Prof. Dr. Vinícius Ruela Pereira Borges

`viniciusrpb@unb.br`

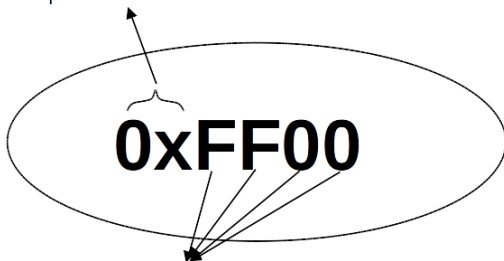
Brasília-DF, 2017

- Introdução e definições
- Operações com ponteiros
- Ponteiros e funções
- Ponteiros e arrays
- Ponteiros e estruturas
- Ponteiro para void

Endereços



0x indica que o número está na base hexadecimal



Cada dígito engloba 4 *bits*. Logo, o número possui **16 bits**


- Endereçamento depende do tipo de processador
- Processador x86: faz endereçamento de memória usando 32 bits!
 - ❶ Endereços com 32 bits: 0xFFA8004C
- Processador x86_64: faz endereçamento de memória usando 48 bits!
 - ❶ Endereços com 48 bits: 0xFFA8004CBBDD

- Do ponto de vista da máquina, na memória estão apenas bits!
- Mas do ponto de vista do programador, o que está na memória depende do tipo de variável associado ao endereço que nos referimos.
- Cada tipo de variável nos fornece:
 - 1 Espaço ocupado na memória
 - 2 Como interpretar o padrão de bits

- Um ponteiro é uma variável que contém um endereço de memória
 - Endereço indica a localização de uma outra variável na memória
 - Proporciona um modo de acesso a uma variável sem referenciá-la

Definição

- Um endereço é a referência que o computador usa para localizar variáveis
- As variáveis ocupam uma posição na memória e seu endereço é o **primeiro byte** ocupado por ela



Endereço	Conteúdo
0xFF7	
0xFF6	A
0xFF5	
0xFF4	
0xFF3	B
0xFF2	
0xFF1	
0xFF0	

Espaço ocupado por tipos

- Fonte: GCC 4.5.0

tipo	tamanho (em bytes)
char	1
int	4
float	4
unsigned	4
double	8
long	8

- **Ponteiros constantes:** arrays alocados estaticamente
 - Ponteiro constante é um endereço, mas não pode ter seu valor alterado
- **Ponteiros variáveis:** tipo de variável que contém o endereço de uma outra variável
 - Lugar para guardar endereços

- Uma variável do tipo ponteiro é declarada como:

`tipo *nome;`

- `nome` é o nome da variável ponteiro
 - `tipo` é qualquer tipo de dados válido na linguagem C
- `tipo` indica o tipo de variável que o ponteiro pode apontar.
 - embora qualquer tipo de ponteiro pode apontar para qualquer endereço de memória

Operador de Endereços (&)

- O operador & retorna o endereço de memória do operando (é um operador unário)

```
1  int *m;  
2  int n;  
3  
4  n = 4;  
5  
6  m = &n;
```

- No trecho de código acima, `m` contém o endereço da memória da variável `n`
- Diz-se que a variável `m` aponta para a variável `n`
- Resumidamente, & significa “o endereço de”

Operador de Endereços (&)

- Para imprimir um endereço, pode-se utilizar o formato %p

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int var = 15;
6      printf("Endereço de var é: %p\n",&var);
7      return 0;
8  }
```

- Obtém-se o endereço na base hexadecimal:

“Endereço de var é: 0x7ffe810fc514”

Operador Indireto (*)

- O operador unário * retorna o conteúdo (ou valor) localizado em um endereço
- É o complemento do operador &
- Pode ser imaginado como “no endereço”
- Ponteiros são sempre inicializados com valor NULL ou 0.
 - Lembre-se: NULL não é um endereço válido

Operador Indireto (*)

```
1  #include <stdio.h>
2
3  int main()
4  {
5      float x = 4.5;
6      float *p;
7
8      p = &x;
9      printf("Endereço de x: %p.\n",&x);
10     printf("Valor de x: %f.\n",x);
11     printf("Endereço apontado por p: %p\n",p);
12     printf("Conteúdo do ponteiro real *p: %f\n",*p);
13
14     return 0;
15 }
```

Endereço de x: 0x7ffd59a86e4c.

Valor de x: 4.500000.

Endereço apontado por p: 0x7ffd59a86e4c

Conteúdo do ponteiro real *p: 4.500000

Exemplos

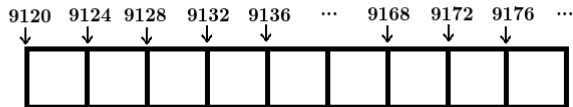
```
1  #include <stdio.h>
2
3  int main()
4  {
5      int a,b;
6      int *c;
7
8      a = 5;
9      b = 3;
10     c = &b;
11
12     printf("Endereço: %p\n",c);
13     printf("Valor: %d\n",*c);
14
15     c = &a;
16
17     printf("Endereço: %p\n",c);
18     printf("Valor: %d\n",*c);
19
20     return 0;
21 }
```


Exemplos

- Início da execução do código-fonte do slide anterior
- Estado da memória:

Memória

Endereços
(Fictícios)



Variáveis

a

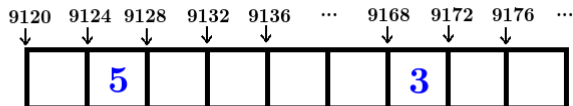
b

**c*

Exemplos

Memória

Endereços
(Fictícios)



Variáveis

a

b

**c*

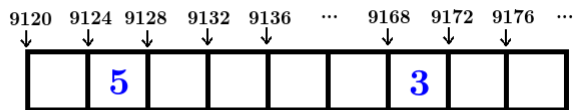
Código-fonte

a = 5;

Exemplos

Memória

Endereços
(Fictícios)



Variáveis

a

b

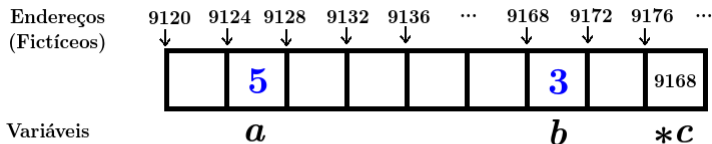
$*c$

Código-fonte

$b = 3;$

Exemplos

Memória



Código-fonte

$c = \&b;$

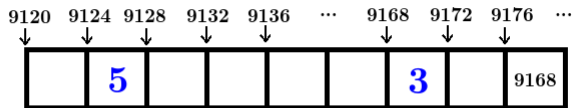
Operador que devolve o endereço na memória do operador, isto é, *b*

A variável *c* aponta para a variável *b*

Exemplos


Memória

Endereços
(Fictícios)



Variáveis

Código-fonte

$c = b;$  No entanto, observe que caso $\&$ seja omitido, a variável c armazenará o endereço "3"

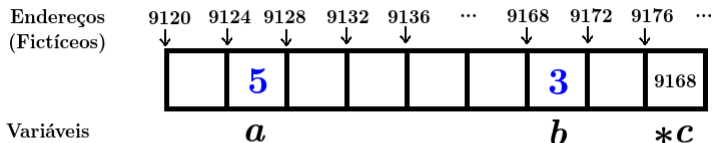
Falha de segmentação

- Por isso, o seguinte erro ocorre em tempo de execução, pois não se sabe o conteúdo do endereço 3

```
-OptiPlex-7040:~$ gcc ponteiros2.c -o ponteiros
ponteiros2.c: In function 'main':
ponteiros2.c:10:7: warning: assignment makes pointer from integer without a cast [-Wint-conversion]
    c = b;
    ^
-OptiPlex-7040:~$ ./ponteiros
Segmentation fault (core dumped)
-OptiPlex-7040:~$
```

Exemplos

Memória



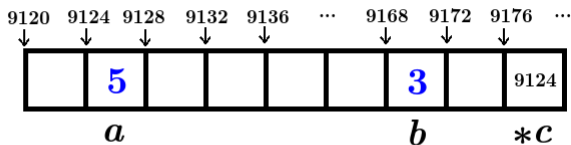
Código-fonte

```
printf("Endereço: %p\n",c); "Endereço: 9168"  
printf("Valor: %d\n",*c);  "Valor: 3"
```

Exemplos

Memória

Endereços
(Fictícios)



Variáveis

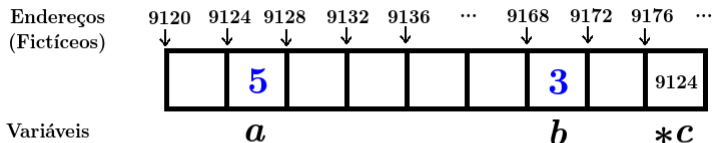
Código-fonte

$c = \&a;$

Agora, a variável *c*
aponta para *a*

Exemplos

Memória



Código-fonte

```
printf("Endereço: %p\n",c); "Endereço: 9124 "  
printf("Valor: %d\n",*c);  "Valor: 5 "
```

Modificação indireta de variável

- Utilizando-se um ponteiro, pode-se alterar indiretamente o conteúdo da variável que ele aponta, isto é, este ponteiro armazena o endereço desta variável

```
1  #include <stdio.h>
2  int main()
3  {
4      int x=5;
5      int *p = &x;
6
7      printf("Valor de x (acesso por *p): %d\n",*p);
8
9      *p = 1;
10
11     printf("Valor de x (acesso por *p): %d\n",*p);
12     return 0;
13 }
```

Valor de x (acesso por *p): 5.

Valor de x (acesso por *p): 1.

Observação 1

- As variáveis ponteiro devem sempre apontar para o tipo de dado correto
 - ponteiro inteiro (`int *`) deve apontar para variável inteira,
 - ponteiro real (`float *`) deve apontar para variável real...
- O código-fonte abaixo compila?

```
1  #include <stdio.h>
2  int main()
3  {
4      int x = 5;
5      double *p = &x;
6
7      printf("Valor de x (acesso por *p): %f\n",*p);
8
9      return 0;
10 }
```

Observação 1

- Compila, mas o compilador emite um *warning*
- Na execução, o programa apresenta um comportamento “inesperado”

```
~$  
~$ gcc vetor_ponteiro.c -o vetor  
vetor_ponteiro.c: In function 'main':  
vetor_ponteiro.c:24:17: warning: initialization from incompatible pointer type [-Wincompatible-pointer-types]  
    double *p = &x;  
                   ^  
~$ ./vetor  
Valor de x (acesso por *p): 0.000000  
~$
```

- Portanto, o compilador permite esse tipo de atribuição, mas não produz o resultado desejado

Observação 2: nunca faça isso

- Utilize & apenas em variáveis declaradas. Nunca faça isso:
 - `&(i+1)`
 - `&10`
- Sempre associe um ponteiro para um endereço. Nunca faça isso:

```
1  #include <stdio.h>
2  int main()
3  {
4      double *p = 5;
5
6      return 0;
7  }
```

Ponteiro para ponteiro

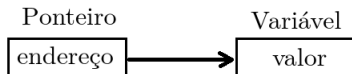
- Um ponteiro para ponteiro (**) é um modo de indireção múltipla
- Ponteiro apontando para outro ponteiro, que aponta para o valor final
- Declaração:

```
tipo **nome;
```

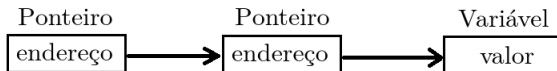
- Exemplos: `int **p;`, `float **f;`

Ponteiro para ponteiro

Indireção simples



Indireção múltipla



- O primeiro ponteiro aponta um segundo ponteiro que aponta para a variável que contém o valor desejado
- Em outras palavras, a variável “**” armazena o endereço de outra variável ponteiro, que contém o endereço de outra variável

Ponteiro para ponteiro

```
1  #include <stdio.h>
2  int main()
3  {
4      int x,*p,**q;
5      x = 5;
6      p = &x;
7      q = &p;
8
9      printf("x = %d    *p = %d    **q = %d\n",x,*p,**q);
10     printf("&x = %p    p = %p    &p = %p    q=%p    &q = %p\n",&x,p,&p,q,&q);
11
12     return 0;
13 }
```

x = 5 *p = 5 **q = 5

&x = 0x7ffc9ac15c54

p = 0x7ffc9ac15c54

&p = 0x7ffc9ac15c58

q=0x7ffc9ac15c58

&q = 0x7ffc9ac15c60

Exemplo: Ponteiro para ponteiro

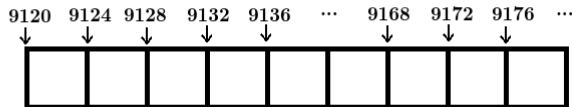
```
1  #include <stdio.h>
2  int main()
3  {
4      int a,*b,**c;
5      a = 5;
6      b = &a;
7      c = &b;
8
9      printf("a = %d    *b = %d    **c = %d\n",a,*b,**c);
10
11     return 0;
12 }
```

Exemplo: Ponteiro para ponteiro

- Início da execução do código-fonte do slide anterior
- Estado da memória:

Memória

Endereços
(Fictícios)



Variáveis

a

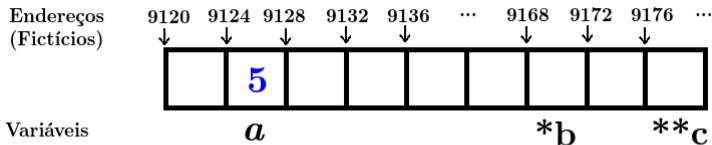
**b*

***c*

Exemplo: Ponteiro para ponteiro

- Estado da memória:

Memória



Código-fonte

```
 $a = 5;$ 
```


Operações com ponteiros

Operações com ponteiros

- Pode-se realizar as seguintes operações com ponteiros:
 - Soma
 - Subtração
 - Operações lógicas ($<$, $>$, $==$, $!=$)

Operações com ponteiros: adição

- Seja p1 um ponteiro para um inteiro e p2 um ponteiro para caractere. As seguintes adições podem ser feitas:

operação	endereço atual	endereço resultante
p1++	2000	2004
p2++	2000	2001
p1 = p1 + 6	2000	2024
p2 = p2 + 12	2000	2012

- Quando um ponteiro é incrementado, ele passa a apontar para a posição de memória do próximo elemento do seu tipo base.

Operações com ponteiros: subtração

- Seja p1 um ponteiro para um inteiro e p2 um ponteiro para caractere. As seguintes subtrações podem ser feitas:

operação	endereço atual	endereço resultante
p1--	2000	1996
p2--	2000	1999
p1 = p1 - 6	2000	1976
p2 = p1 - 12	2000	1988

- Quando um ponteiro é decrementado, ele passa a apontar para a posição de memória anterior do seu tipo base.

Operações com ponteiros: comparações

- Ponteiros podem ser comparados em uma expressão relacional
- Sejam p e q dois ponteiros. A expressão abaixo compara os endereços contidos em ambos:

```
if(p < q)
    printf("p aponta para um endereco mais
baixo que q\n");
```

Operações com ponteiros: exercício

- Responda o que será impresso após a execução do programa abaixo:

```
1  #include <stdio.h>
2  int main()
3  {
4      int a = 1, b = 2;
5      int *pa, *pb;
6      pa = &a;
7      pb = &b;
8
9      if( pa < pb )
10     {
11         printf("pa - pb = %lu\n", pb-pa);
12     }
13     else
14     {
15         printf("pa - pb = %lu\n", pa-pb);
16     }
17     printf("pa = %p, *pa = %d, &pa = %p\n", pa, *pa, &pa);
18     printf("pb = %p, *pb = %d, &pb = %p\n", pb, *pb, &pb);
19
20     pa++;
21
22     printf("pa = %p, *pa = %d, &pa = %p\n", pa, *pa, &pa);
23
24     pb = pb + 3;
25
26     printf("pb = %p, *pb = %d, &pb = %p\n", pb, *pb, &pb);
27     printf("pb - pa = %lu\n", pb - pa);
28     return 0;
29 }
```

Ponteiros e funções

- Na linguagem C, argumentos são passados para funções usando **Chamada por Valor**
- Basicamente, faz-se uma cópia dos argumentos passados para serem usados dentro da função
- Isso pode causar duas restrições:
 - 1 Memória e tempo de processamento extra são necessários para realizar essa cópia
 - 2 Alterações aos argumentos são feitos localmente, não são visíveis fora da função

- **Estudo de caso:** Como implementar uma função que realiza a troca dos valores de duas variáveis?
- Considere o programa a seguir:

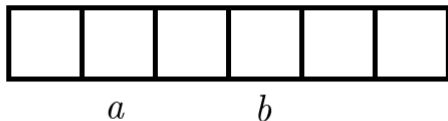
```
1  #include <stdio.h>
2  int main()
3  {
4      int a=3,b=2;
5
6      printf("Antes\nA = %d, B = %d\n",a,b);
7
8      troca(a,b); // Função que troca os valores das
9                  // variáveis a e b
10
11     printf("Depois\nA = %d, B = %d\n",a,b);
12     return 0;
13 }
```

- Como implementar uma função que realiza a troca dos valores de duas variáveis?

```
1 void troca(int x, int y)
2 {
3     int aux;
4
5     aux = x;
6     x = y;
7     y = aux;
8
9 }
```

- **Execução do programa:** início.

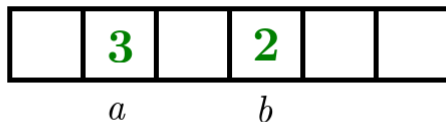
Memória



na função `main()`

- **Execução do programa:** faz-se $a = 3$ e $b = 2$.

Memória

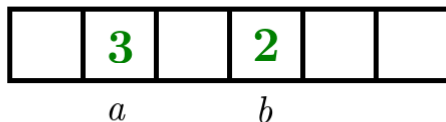


na função `main()`

Ponteiros e funções: exemplo

- **Execução do programa:** Imprime os valores de *a* e *b*.

Memória



na função `main()`

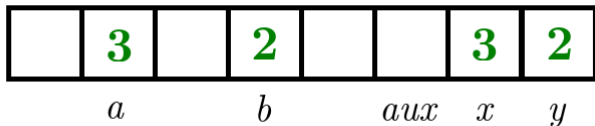
Antes

`A = 3, B = 2`

Ponteiros e funções: exemplo

- **Execução do programa:** Chama a função troca, em que associa-se a variável *a* com a variável *x* e a variável *b* com a variável *y*.

Memória

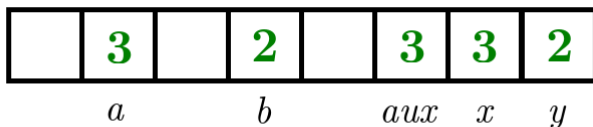


chama a função troca($x = a$, $y = b$)

Ponteiros e funções: exemplo

- **Execução do programa:** Na função troca, faz-se $\text{aux} = x$;

Memória



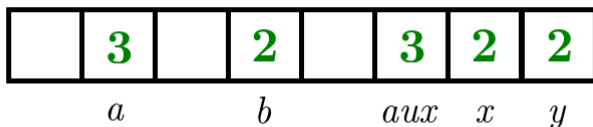
chama a função troca($x = a$, $y = b$)

$\text{aux} = x$;

Ponteiros e funções: exemplo

- **Execução do programa:** Na função troca, faz-se $x = y$;

Memória



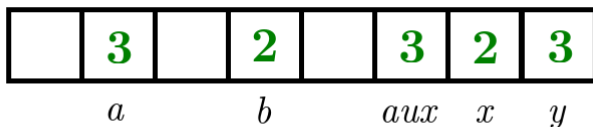
chama a função troca($x = a$, $y = b$)

$x = y$;

Ponteiros e funções: exemplo

- **Execução do programa:** Na função troca, faz-se $y = aux;$.

Memória

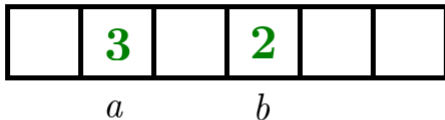


chama a função troca($x = a$, $y = b$)
 $y = aux;$

Ponteiros e funções: exemplo

- **Execução do programa:** Função troca é encerrada.
Imprime os valores de a e b.

Memória



na função `main()`

`função troca()` é encerrada

A TROCA NÃO FOI EFETUADA!

Depois

A = 3, B = 2

- **Solução utilizando ponteiros:**

- Ao invés de passar apenas os valores de a e b, vamos passar seus endereços

```
1  #include <stdio.h>
2  int main()
3  {
4      int a=3,b=2;
5
6      printf("Antes\nA = %d, B = %d\n",a,b);
7
8      troca(&a,&b); // Passamos o endereço das variáveis a e b
9                  // pois seus endereços são permanentes até
10                 // o final da execução deste programa
11
12      printf("Depois\nA = %d, B = %d\n",a,b);
13      return 0;
14 }
```


- Como a função `troca` agora deve receber dois endereços, substituímos as variáveis inteiras por ponteiros inteiros
 - Assim podemos acessar os valores a partir dos endereços e efetuar a troca

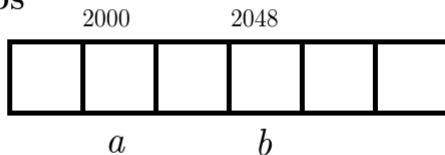
```
1 void troca(int *x, int *y)
2 {
3     int aux;
4
5     aux = *x;
6     *x = *y;
7     *y = aux;
8
9 }
```

Ponteiros e funções: exemplo

- **Execução do programa:** início.

Memória

ENDEREÇOS



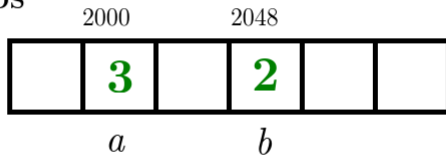
na função `main()`

Ponteiros e funções: exemplo

- **Execução do programa:** faz-se $a = 3$ e $b = 2$.

Memória

ENDEREÇOS



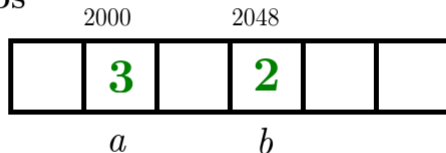
na função `main()`

Ponteiros e funções: exemplo

- **Execução do programa:** Imprime os valores de *a* e *b*.

Memória

ENDEREÇOS



na função `main()`

Antes

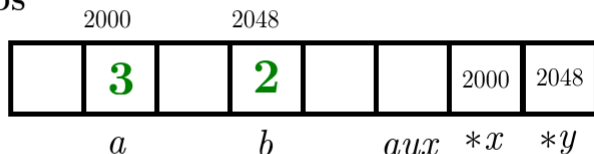
`A = 3, B = 2`

Ponteiros e funções: exemplo

- **Execução do programa:** Chama a função troca, em que os endereços de a e b são passados como argumentos.

Memória

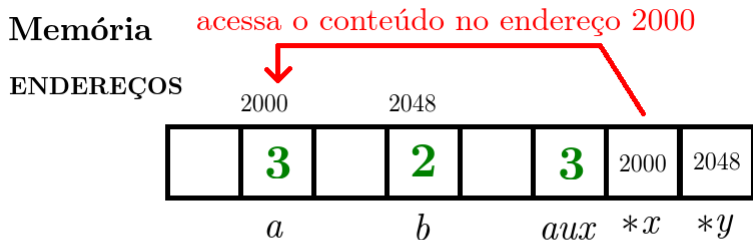
ENDEREÇOS



chama a função troca ($x = \&a$, $y = \&b$)

Ponteiros e funções: exemplo

- Execução do programa: Faz-se `aux = *x`.

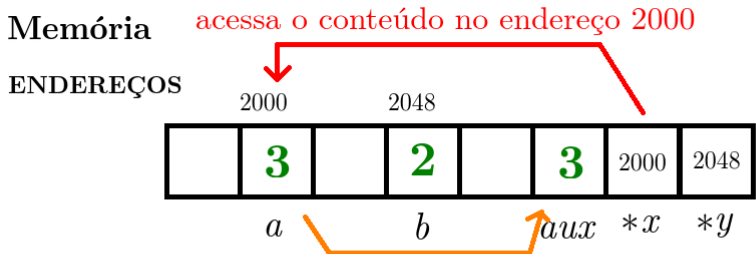


chama a função troca (`x = &a, y = &b`)

`aux = *x;`

Ponteiros e funções: exemplo

- **Execução do programa:** Faz-se $\text{aux} = *x$.



chama a função troca ($x = \&a$, $y = \&b$)

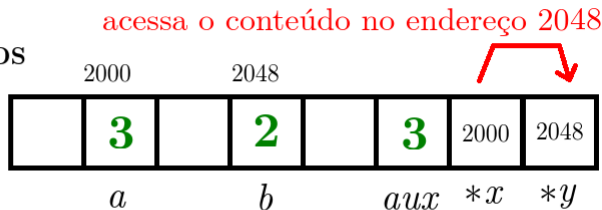
$\text{aux} = *x$; Atribui para aux o conteúdo do endereço apontado por x

Ponteiros e funções: exemplo

- Execução do programa: Faz-se $*x = *y$.

Memória

ENDEREÇOS



chama a função troca ($x = \&a$, $y = \&b$)

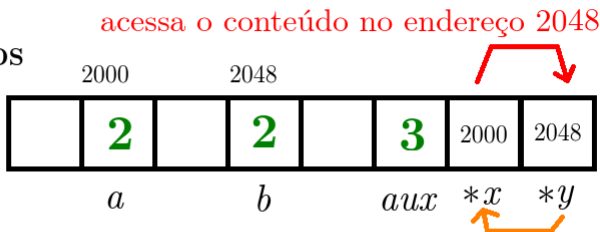
$*x = *y;$

Ponteiros e funções: exemplo

- **Execução do programa:** Faz-se $*x = *y$.

Memória

ENDEREÇOS



chama a função troca ($x = \&a$, $y = \&b$)

$*x = *y;$

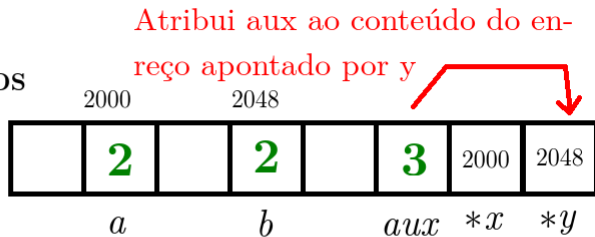
o conteúdo do endereço 2048
é atribuído ao conteúdo do
endereço 2000

Ponteiros e funções: exemplo

- **Execução do programa:** Faz-se $*y = aux$.

Memória

ENDEREÇOS



chama a função troca ($x = \&a$, $y = \&b$)

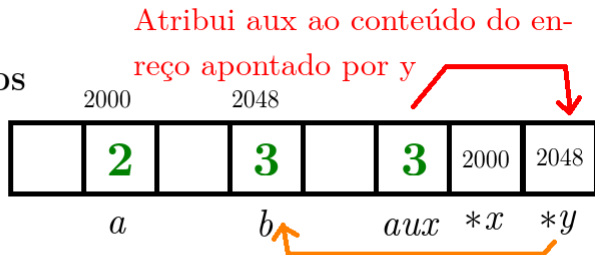
$*y = aux$;

Ponteiros e funções: exemplo

- Execução do programa: Faz-se $*y = aux$.

Memória

ENDEREÇOS



chama a função troca ($x = \&a$, $y = \&b$)

$*y = aux$;

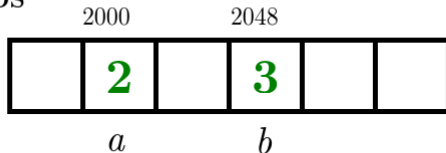
Atualiza o conteúdo de b para o valor contido em aux

Ponteiros e funções: exemplo

- **Execução do programa:** Encerra-se a função `troca(...)` e imprime os valores de `a` e `b`.

Memória

ENDEREÇOS



na função `main()`

função `troca` é encerrada!

Depois

`A = 2, B = 3`

- Para passar ponteiros como parâmetros (argumentos) de funções, passe apenas o endereço armazenado pelo ponteiro

```
1  #include <stdio.h>
2
3  void imprimePonteiro(int *p)
4  {
5      printf("Valor no endereco: %d\n",*p);
6  }
7
8  int main()
9  {
10     int a,*pont;
11     a = 6;
12     pont = &a;
13
14     imprimePonteiro(pont); // endereço armazenado em pont
15                           // é passado como parametro
16     return 0;
17 }
```

- Os tipos passados como argumentos e recebidos na função devem ser os mesmos:

```
1  #include <stdio.h>
2
3  char * imprimePonteiro()
4  {
5      printf("Valor no endereco: %d\n",*p);
6  }
7
8  int main()
9  {
10     int a,*pont;
11     a = 6;
12     pont = &a;
13
14     imprimePonteiro(pont); // endereço armazenado em pont
15                           // é passado como parametro
16
17     return 0;
18 }
```

- A passagem de ponteiros como parâmetros de funções viabiliza o acesso aos argumentos originais passados
 - Possibilita “retornar” valores nos argumentos das funções
 - Evita cópia de argumentos muito grandes
- As formas de passagens de parâmetros vistas anteriormente são chamadas de **Chamada por Referência**
 - Nas funções, ocorrem cópias dos ponteiros, que indicam os endereços dos argumentos originais

Ponteiros e arrays

- Tradicionalmente, um vetor de inteiros pode ser impresso como:

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int vetor[] = {6,2,4,0,1};
6      int i;
7
8      for(i = 0; i < 4; i++)
9      {
10         printf("|%d",vetor[i]);
11     }
12     printf("|%d|\n",vetor[i]);
13
14     return 0;
15 }
```

- Como implementar a operação de imprimir um vetor de inteiros utilizando ponteiros?
- Utilizando a aritmética de ponteiros e conceitos fundamentais de alocação estática de arrays, pode-se trabalhar com vetores utilizando ponteiros
 - Na alocação estática, o espaço em memória de variáveis (incluindo arrays) é determinado em **tempo de compilação**.
 - Além disso, os dados são organizados na memória, de forma linear e sequencial

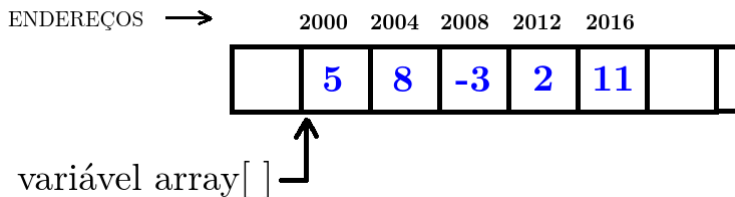
Ponteiros e arrays

- Alocação estática de um vetor:

```
int array[] = {5,8,-3,2,11};
```

- A variável `array[]` armazena o endereço da primeira posição, no caso abaixo, `array = 2000`:

Memória (Fictícia)



- Existe uma estreita relação entre ponteiros e matrizes
- O nome de uma matriz é um endereço, ou seja, um ponteiro
- Utilizando ponteiros em linguagem C, qualquer operação que possa ser feita com índice de matrizes
 - Uma variável matriz é um ponteiro constante

Ponteiros e arrays

- O acesso ao elemento de um vetor dado por índice (`nomeVariavelVetor[índice]`) utilizando ponteiros ocorre como:

`*(nomeVariavelVetor + índice)`

- Versão utilizando ponteiros:

```
1  int main()
2  {
3      int vetor[] = {6,2,4,0,1};
4      int i;
5
6      for(i = 0; i < 4; i++)
7      {
8          printf("|%d",*(vetor+i));
9      }
10     printf("|%d\\n",*(vetor+i));
11
12     return 0;
13 }
```

- Na função abaixo, a declaração do vetor de inteiros vetor[] é equivalente a *vetor na função imprimeVetor():

```
1  #include <stdio.h>
2
3  void imprimeVetor(int *vetor, int n)
4  {
5      int i;
6      for(i = 0; i < n-1; i++)
7      {
8          printf("%d",vetor[i]);
9      }
10     printf("%d\\n",vetor[i]);
11 }
12
13 int main()
14 {
15     int vv[] = {6,2,4,0,1};
16
17     imprimeVetor(vv,5);
18
19     return 0;
20 }
```

Ponteiros e arrays

- Reescrevendo a função anterior `imprimeVetor()` utilizando apenas ponteiros, tem-se:

```
1  #include <stdio.h>
2
3  void imprimeVetor_pont(int *vetor, int n)
4  {
5      int i;
6      for(i = 0; i < n-1; i++)
7      {
8          printf("|%d",*(vetor+i));
9      }
10     printf("|%d|\n",*(vetor+(n-1)));
11 }
12
13 int main()
14 {
15     int vv[] = {6,2,4,0,1};
16
17     imprimeVetor_pont(vv,5);
18
19     return 0;
20 }
```

Ponteiros e arrays

- O endereço de um elemento de um array pode ser referenciado de duas formas:
 - 1 em notação ponteiro (`nums+d`)
 - 2 em notação de array (`& nums[d]`)
- Considerando um vetor de inteiros, no endereço 2000:

```
int nums[] = { 5,3,1,2,4 }  
& nums[2] == (nums+2) == 2008  
nums[2] == *(nums+2) == 1
```


- Uma string pode ser declarada utilizando ponteiros:

```
1  #include <stdio.h>
2
3  int main()
4  {
5      char *str = "Uma string utilizando ponteiros";
6
7      printf("%s\n",str);
8
9      return 0;
10 }
```

Ponteiros e arrays: cuidado!

- **Cuidado:** Ao imprimir uma string utilizando apenas ponteiros, evite fazer:

```
1  #include <stdio.h>
2
3  void imprimeString(char *p)
4  {
5      while(*p != '\0')
6      {
7          printf("%c",*p);
8          p++;
9      }
10     printf("\n");
11 }
12
13 int main()
14 {
15     char *p = "Uma string utilizando ponteiros";
16
17     imprimeString(p);
18
19     return 0;
20 }
```

- Pois perde-se o endereço do primeiro caractere da string `str`

Ponteiros e registros

- Seja a seguinte estrutura:

```
1  #include <stdio.h>
2
3  struct produto
4  {
5      int ano;
6      float peso;
7      char tipo;
8  };
9
10 typedef struct produto Item;
11
12 int main()
13 {
14     Item a;
15
16     a.ano = 1997;
17     a.peso = 56.1;
18     a.tipo = 'P';
19
20     printf("Produto:\nAno: %d\nPeso: %.2f\nTipo: %c\n",a.ano,a.peso,a.tipo
21           );
22     return 0;
23 }
```

- Pode-se declarar um ponteiro para uma estrutura conforme:

```
Item a;
```

```
Item *pont_a;
```

```
pont_a = &a;
```

- E realizar o acesso indireto:

```
(*pont_a).ano = 1998;
```

```
(*pont_a).peso = 39.3;
```

```
(*pont_a).tipo = 'N';
```

- A notação do tipo `(*pont_a).ano` é confusa, de forma que a linguagem C define um operador adicional (`->`) para acessar membros de estruturas através de ponteiros
- O operador `->` substitui o operador `.` no caso da utilização de um ponteiro para uma estrutura (`struct`)

`(*pont_a).ano = 1998;`

- é equivalente a

`pont_a->ano = 1998;`

Ponteiros e registros

- O programa abaixo imprime:

```
1  int main()
2  {
3      Item a;
4      Item *pont_a;
5
6      pont_a = &a;
7
8      pont_a->ano = 1997;
9      pont_a->peso = 56.1;
10     pont_a->tipo = 'P';
11
12     printf("Produto:\nAno: %d\nPeso: %.2f\nTipo: %c\n", pont_a->ano, pont_a->peso,
13           pont_a->tipo);
14
15     return 0;
16 }
```

Produto:

Ano: 1997

Peso: 56.10

Tipo: P

Ponteiros e registros

- Leitura de dados utilizando ponteiro para estrutura:

```
1  #include <stdio.h>
2
3  /* .. declaração da mesma struct produto .. */
4
5  int main()
6  {
7      Item a;
8      Item *pont_a;
9
10     pont_a = &a; // pont_a aponta para a variavel da estrutura a
11
12     printf("Digite o ano: ");
13     scanf("%d",&pont_a->ano);
14     printf("Digite o peso: ");
15     scanf("%f",&pont_a->peso);
16     scanf("%*c"); // Ler '\n' após o float digitado anteriormente
17     printf("Digite o tipo: ");
18     scanf("%c",&pont_a->tipo);
19
20     printf("Produto:\nAno: %d\nPeso: %.2f\nTipo: %c\n",pont_a->ano,pont_a->peso,
21           pont_a->tipo);
22
23     return 0;
24 }
```


Ponteiro para void

Ponteiro para void (void *)

- O `void *` é um ponteiro genérico de dados
 - Não há tipo de dados associado
- Ocupa 8 bytes na memória
- Armazena endereços de qualquer tipo e pode ser “casteado” para qualquer tipo
- Importante nas funções de alocação dinâmica de memória

Ponteiro para void (void *)

- Um ponteiro para void é declarado da seguinte maneira:
`void *nome;`
- em que nome é o nome da variável ponteiro void.
- **“Casting”**: seja *ponteiro um ponteiro para void. O cast de um ponteiro para void para um tipo de dados é feito conforme:

```
tipo nomeVariavel = * (tipo *) ponteiro;
```

- **“Casting” correto**

```
void funcao(void *ponteiro) {  
    int inteiro = * (int *) ponteiro;  
}
```

- **“Casting” correto**

```
void funcao(void *ponteiro) {  
    int *pontInteiro = (int *) ponteiro;  
}
```

- **“Casting” incorreto**

```
void funcao(void *ponteiro) {  
    int inteiro = *ponteiro; // errado!!  
}
```

- **“Casting” incorreto**

```
void funcao(void *ponteiro) {  
    int *inteiro = *ponteiro; // errado!!  
}
```

Ponteiro para void (void *)

- Exemplo:

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int a =10;
6      int *pont_int;
7      char b = 'x';
8      void *p = &a;  // p aponta para a
9
10     printf("%d",*(int *) p);
11
12     p = &b;  // p aponta para b
13
14     printf("%c",*(char *) p);
15
16     pont_int = (int *) p;
17
18     printf("%d",*(int *) p);
19
20     return 0;
21 }
```

- SCHILDT, H. C Completo e Total. 3a edição. Pearson Makron Books, 2006.
- TENENBAUM, Aaron M.; LANGSAM, Yedidiah; AUGENSTEIN, Moshe J. Estruturas de dados usando C. Pearson Makron Books, 2004.