FUNÇÕES (continuação): Passagem de parâmetros

Introdução à Programação

Profª. Giorgia Mattos – giorgiamattos@gmail.com

Tipos de Passagem de Parâmetros

- Passagem por valor:
  - O parâmetro formal recebe uma copia do parâmetro real
  - Qualquer modificação dentro da função só altera o parâmetro formal

- Passagem por referência:
  - O parâmetro real é uma variável
  - O parâmetro real se torna o parâmetro formal (mesmo espaço de memória)
  - Modificações são feitas também no parâmetro real
  - Não existe em C!!

Modos de Parâmetros

- Os parâmetros são a forma que as partes do programas tem para se comunicar com as funções

- **Parâmetros de entrada** informam a função os dados que ela deve utilizar para executar sua funcionalidade

Todos os parâmetros utilizados até agora foram de entrada: as funções apenas utilizaram os valores passados, apenas fizeram a *leitura* deles para efetuarem suas funcionalidades.

Modos de Parâmetros

- Os parâmetros são a forma que as partes do programas tem para se comunicar com as funções

- **Parâmetros de entrada** informam a função os dados que ela deve utilizar para executar sua funcionalidade

- Alguns **parâmetros de saída** podem ter os valores modificados pela função como parte do resultado.
  - Funcionam como um mecanismo a mais para a produção de valores pelas funções, além do valor de retorno

No entanto, também é possível criar parâmetros de saída: parâmetros que terão seus valores *alterados* pela função – a função fará uma *escrita* de novos valores nos parâmetros.

Modos de Parâmetros

- Os parâmetros são a forma que as partes do programas tem para se comunicar com as funções
- **Parâmetros de entrada** informam a função os dados que ela deve utilizar para executar sua funcionalidade
- Alguns **parâmetros de saída** podem ter os valores modificados pela função como parte do resultado.
  - Funcionam como um mecanismo a mais para a produção de valores pelas funções, além do valor de retorno
- Dizemos que um parâmetro é de **entrada e saída** quando a função tanto utiliza quanto altera o valor de um parâmetro

Exemplo de parâmetros de entrada e saída.

```c
#include <stdio.h>

void Troca(int x, int y){
        int aux = x;

        x = y;
        y = aux;
}

int main(void){
        int a = 3, b = 5;

        printf("antes: a=%d, b=%d\n", a, b);
        Troca(a, b);
        printf("depois: a=%d, b=%d\n", a, b);
        return 0;
}
```

A intenção do programador era que os **parâmetros** da função fossem de **entrada e saída.**

Exemplo de parâmetros de entrada e saída.

```
#include <stdio.h>

void Troca(int x, int y){
        int aux = x;

        x = y;
        y = aux;
}


int main(void){
        int a = 3, b = 5;

        printf("antes: a=%d, b=%d\n", a, b);
        Troca(a, b);
        printf("depois: a=%d, b=%d\n", a, b);
        return 0;
}
```

O que faria com que os valores das variáveis ***a*** e ***b***, parâmetros reais, fossem trocados após a execução da função.

Exemplo de parâmetros de entrada e saída.

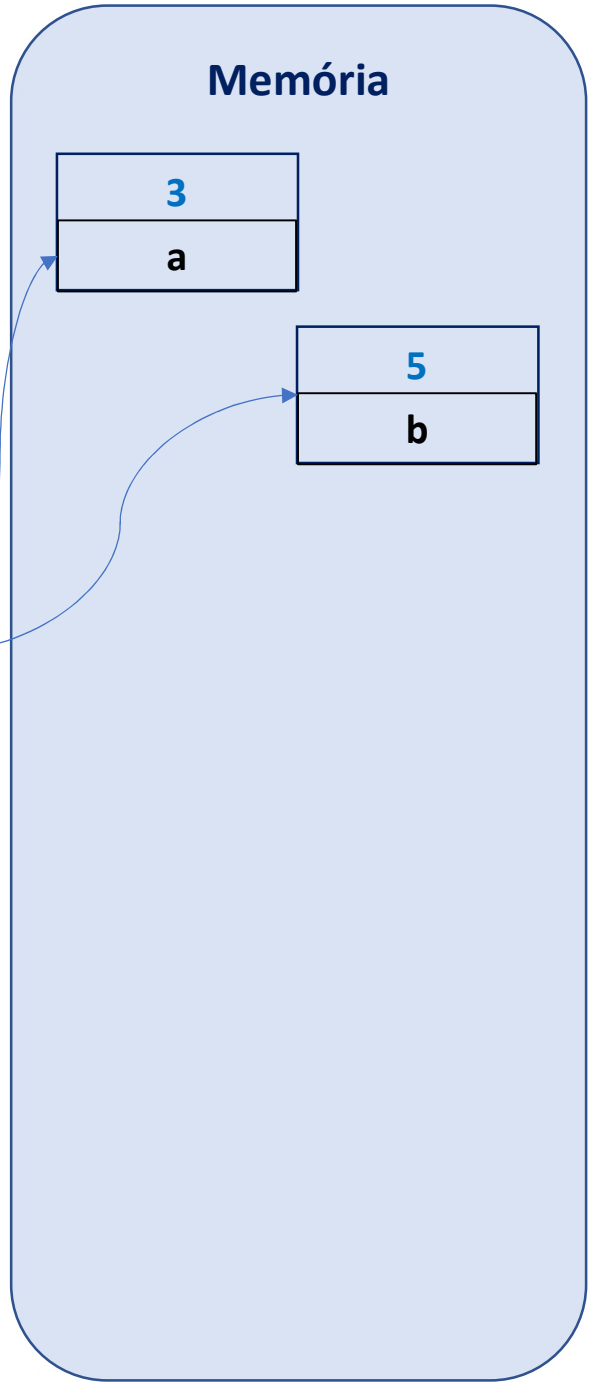
```
#include <stdio.h>

void Troca(int x, int y){
        int aux = x;

        x = y;
        y = aux;
}


int main(void){
        int a = 3, b = 5;

        printf("antes: a=%d, b=%d\n", a, b);
        Troca(a, b);
        printf("depois: a=%d, b=%d\n", a, b);
        return 0;
}
```

Mas não é o que ocorre e os dois **printfs** mostram os mesmos valores.

Isso porque a **passagem de parâmetros em C é sempre por valor**, ou seja...

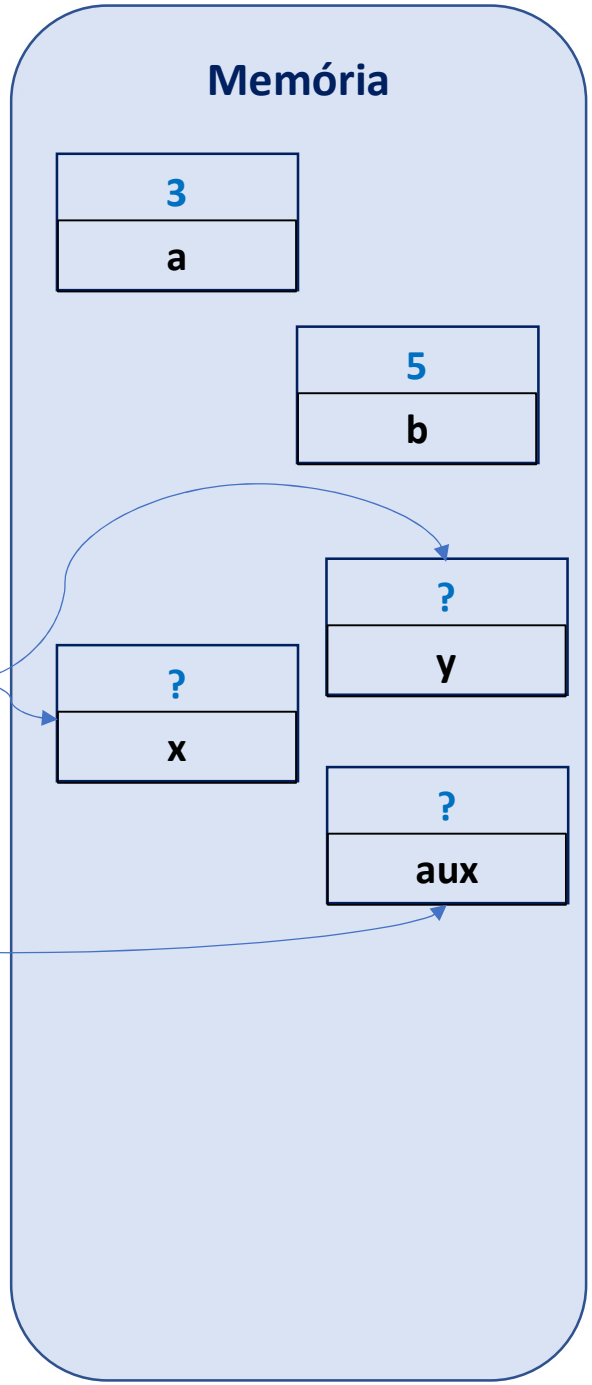
```c
#include <stdio.h>

void Troca(int x, int y){
        int aux = x;

        x = y;
        y = aux;
}

int main(void){
        int a = 3, b = 5;

        printf("antes: a=%d, b=%d\n", a, b);
        Troca(a, b);
        printf("depois: a=%d, b=%d\n", a, b);
        return 0;
}
```

Ao chegar no primeiro ***printf***, a memória do computador apresentará as variáveis **a** e **b** alocadas com os valores 3 e 5



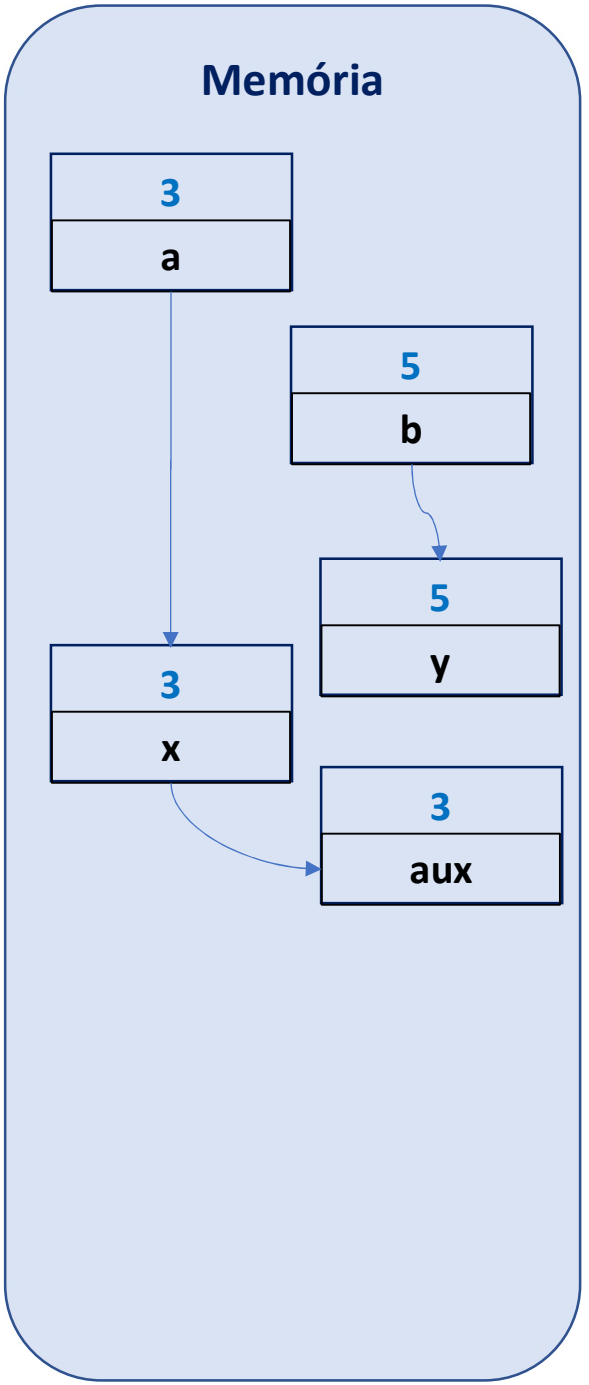
```c
#include <stdio.h>

void Troca(int x, int y){
        int aux = x;

        x = y;
        y = aux;
}

int main(void){
        int a = 3, b = 5;

        printf("antes: a=%d, b=%d\n", a, b);
        Troca(a, b);
        printf("depois: a=%d, b=%d\n", a, b);
        return 0;
}
```

A preparação para a execução da função ***Troca*** alocará os parâmetros e as variáveis de duração automática do bloco da função



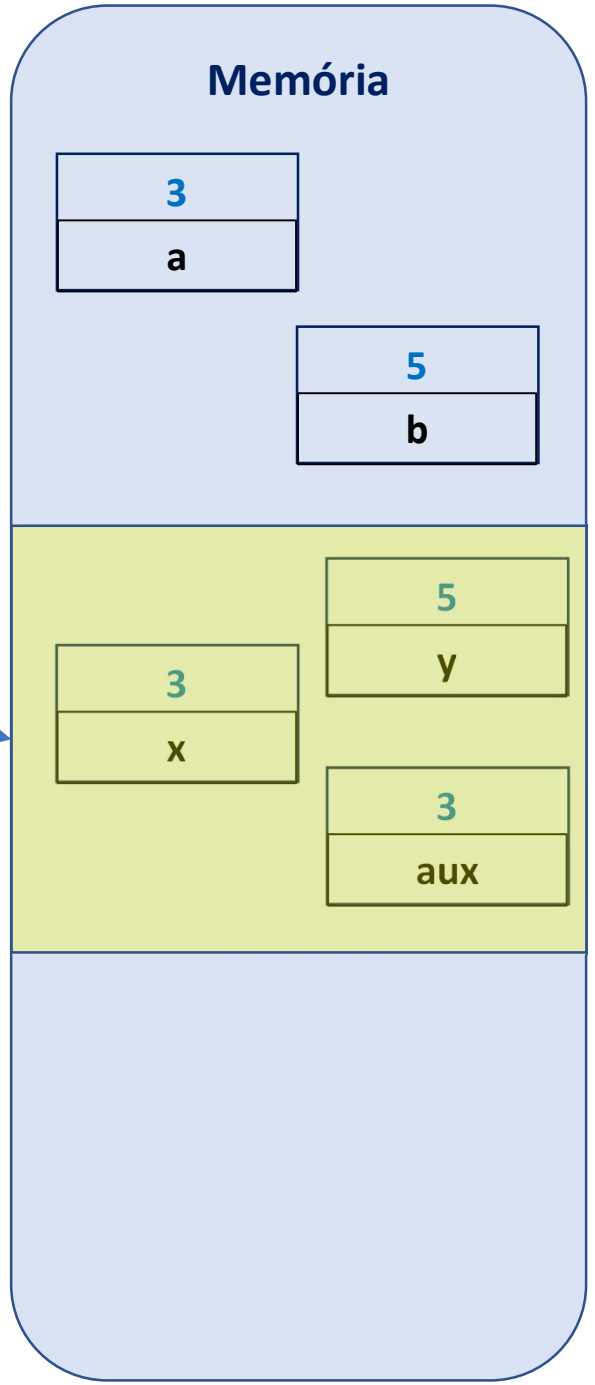
```c
#include <stdio.h>

void Troca(int x, int y){
        int aux = x;

        x = y;
        y = aux;
}

int main(void){
        int a = 3, b = 5;

        printf("antes: a=%d, b=%d\n", a, b);
        Troca(a, b);
        printf("depois: a=%d, b=%d\n", a, b);
        return 0;
}
```

Os parâmetros formais recebem uma cópia dos valores dos parâmetros reais.
A inicialização de aux faz ela receber uma cópia do parâmetro x



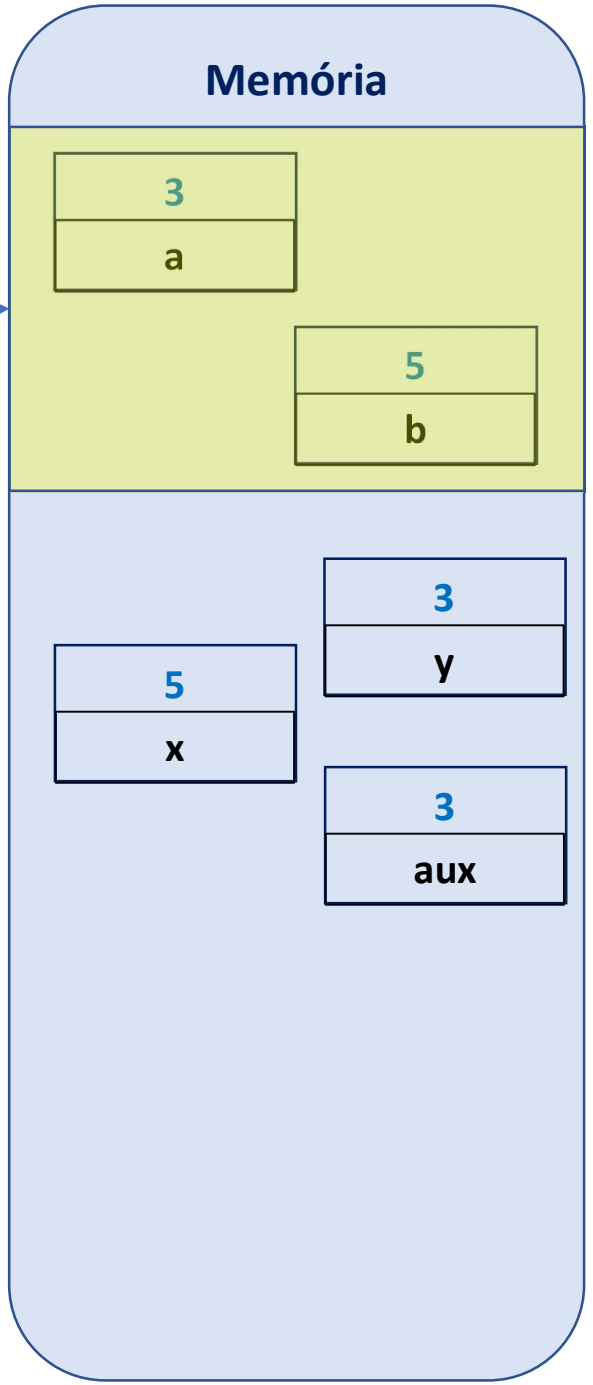
```c
#include <stdio.h>

void Troca(int x, int y){
    int aux = x;

    x = y;
    y = aux;
}

int main(void){
    int a = 3, b = 5;

    printf("antes: a=%d, b=%d\n", a, b);
    Troca(a, b);
    printf("depois: a=%d, b=%d\n", a, b);
    return 0;
}
```

Agora, toda a manipulação de valores ocorrerá apenas nessas variáveis na memória.



```c
#include <stdio.h>

void Troca(int x, int y){
    int aux = x;

    x = y;
    y = aux;
}

int main(void){
    int a = 3, b = 5;

    printf("antes: a=%d, b=%d\n", a, b);
    Troca(a, b);
    printf("depois: a=%d, b=%d\n", a, b);
    return 0;
}
```

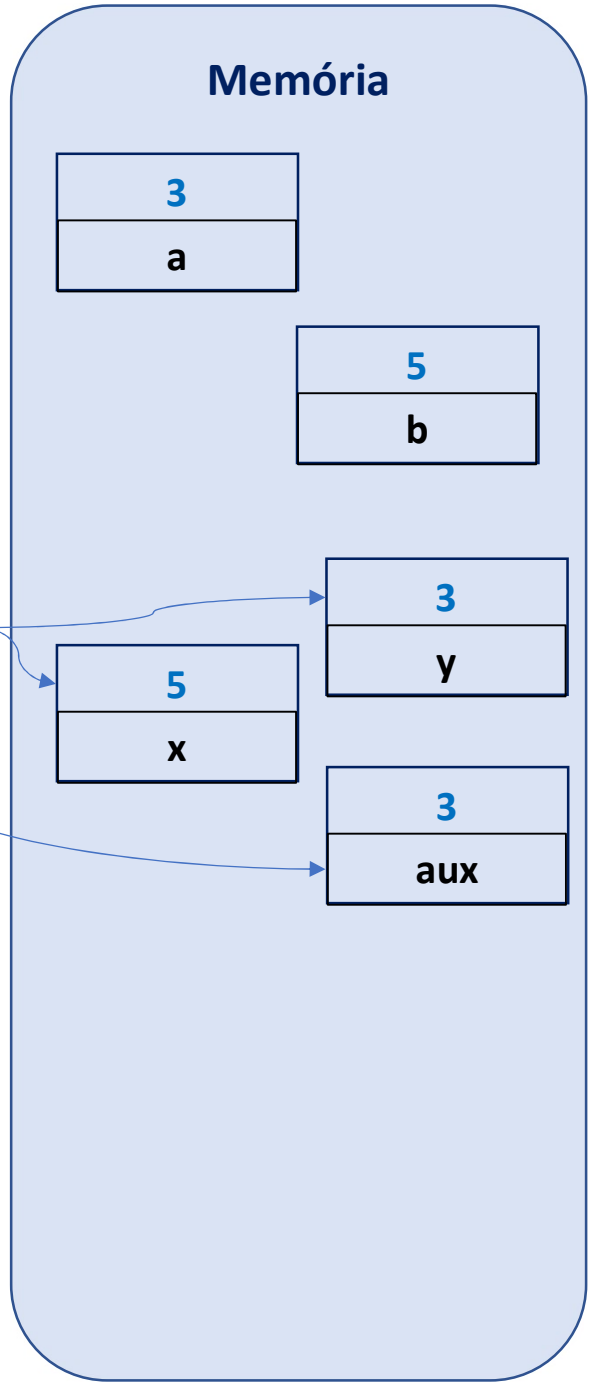
As variáveis ***a*** e ***b*** permanecem **inalteradas**. Afinal, elas nem podem ser acessadas no *escopo* da função ***Troca***.



```c
#include <stdio.h>

void Troca(int x, int y){
        int aux = x;

        x = y;
        y = aux;
}
```

Ao final da execução de ***Troca***, as variáveis de duração automática e parâmetros serão liberados.

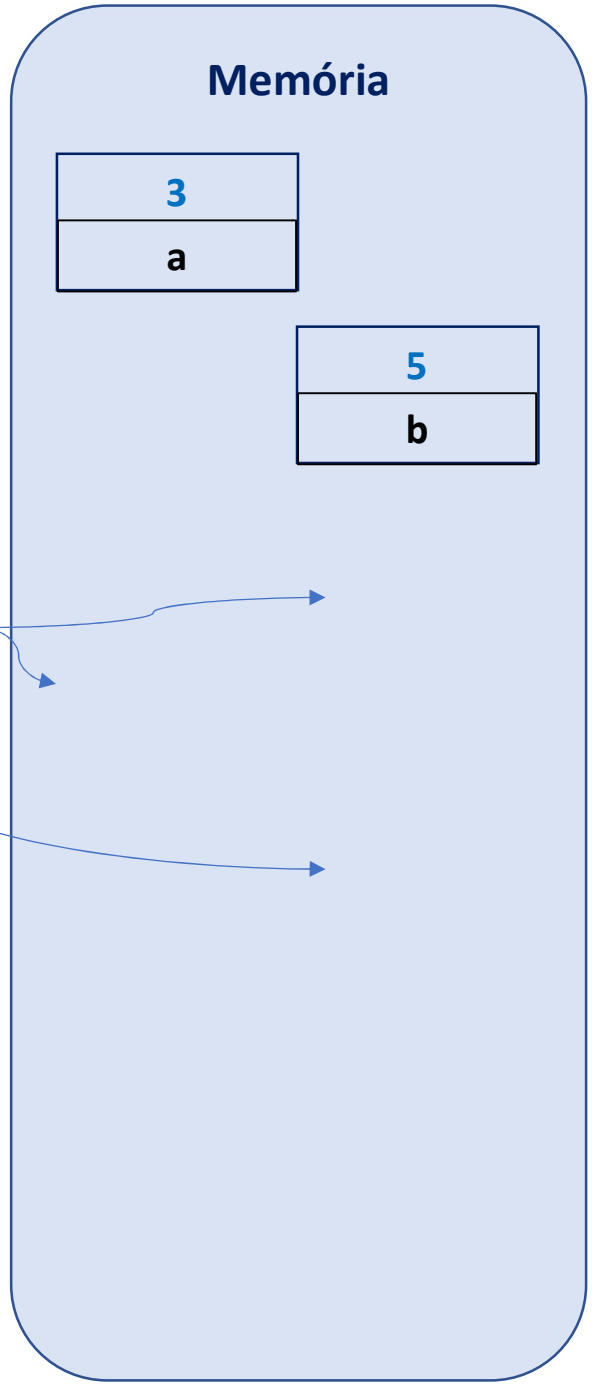
```c
int main(void){
        int a = 3, b = 5;

        printf("antes: a=%d, b=%d\n", a, b);
        Troca(a, b);
        printf("depois: a=%d, b=%d\n", a, b);
        return 0;
}
```



```c
#include <stdio.h>

void Troca(int x, int y){
        int aux = x;

        x = y;
        y = aux;
}
```

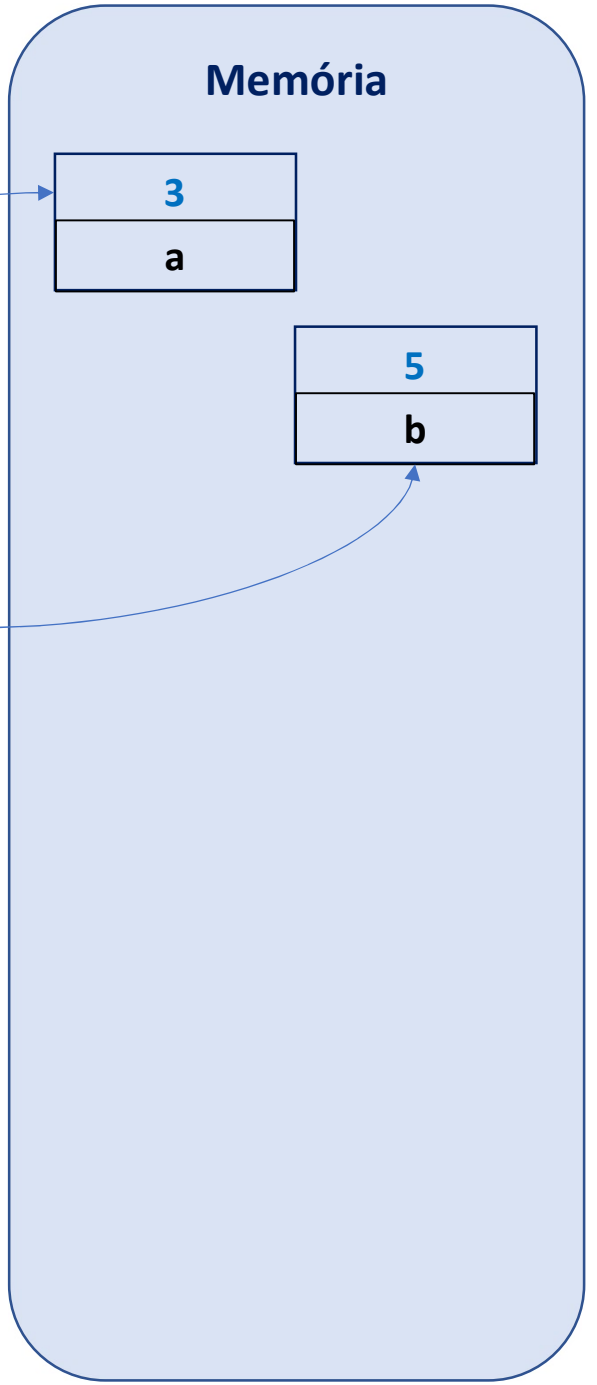
Ao final da execução de ***Troca***, as variáveis de duração automática e parâmetros serão liberados.

```c
int main(void){
        int a = 3, b = 5;

        printf("antes: a=%d, b=%d\n", a, b);
        Troca(a, b);
        printf("depois: a=%d, b=%d\n", a, b);
        return 0;
}
```



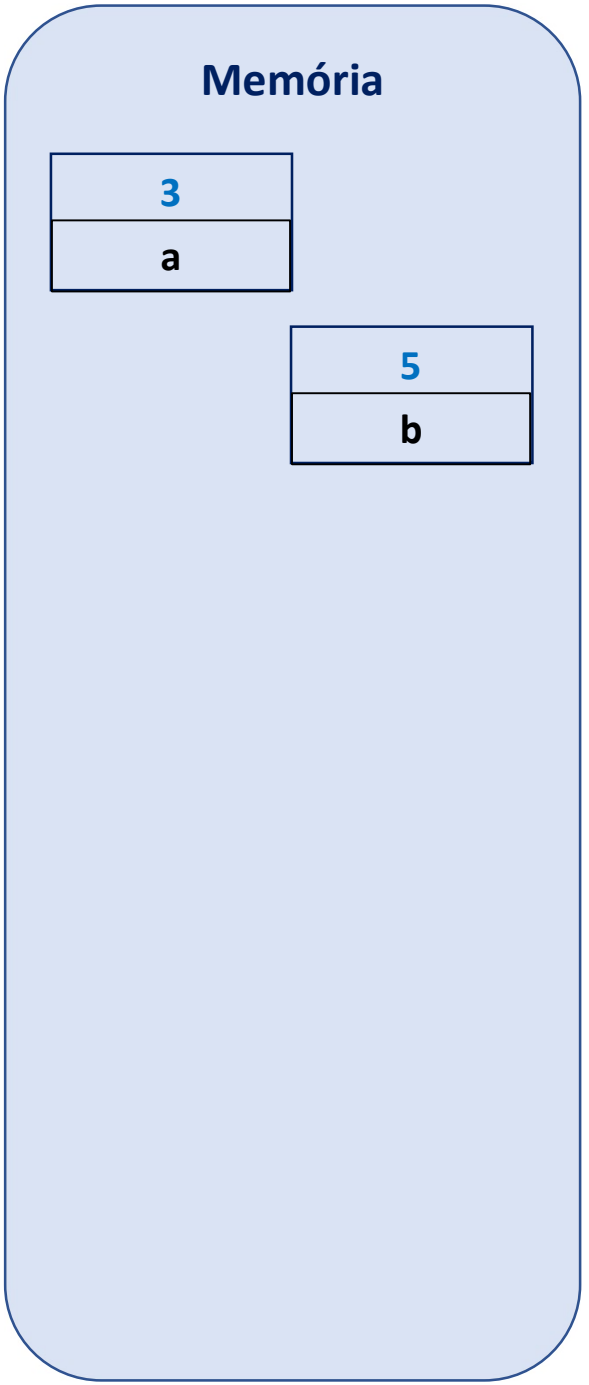
```c
#include <stdio.h>

void Troca(int x, int y){
        int aux = x;

        x = y;
        y = aux;
}

int main(void){
        int a = 3, b = 5;

        printf("antes: a=%d, b=%d\n", a, b);
        Troca(a, b);
        printf("depois: a=%d, b=%d\n", a, b);
        return 0;
}
```

E o segundo ***printf*** imprime mais uma vez os valores de ***a*** e ***b***, que não foram alterados.



```c
#include <stdio.h>

void Troca(int x, int y){
        int aux = x;

        x = y;
        y = aux;
}

int main(void){
        int a = 3, b = 5;

        printf("antes: a=%d, b=%d\n", a, b);
        Troca(a, b);
        printf("depois: a=%d, b=%d\n", a, b);
        return 0;
}
```

Se a passagem de parâmetros fosse por ***referência***, os parâmetros formais ocupariam o mesmo espaço de memória que os parâmetros reais.

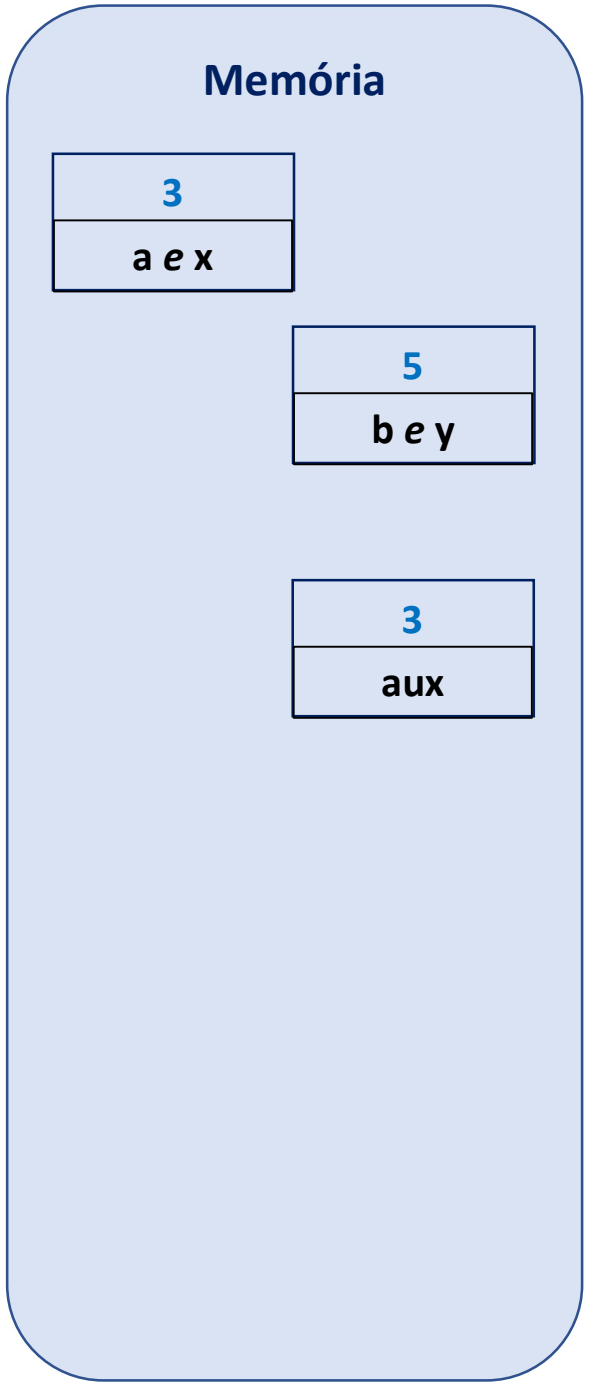


```c
#include <stdio.h>

void Troca(int x, int y){
        int aux = x;

        x = y;
        y = aux;
}

int main(void){
        int a = 3, b = 5;

        printf("antes: a=%d, b=%d\n", a, b);
        Troca(a, b);
        printf("depois: a=%d, b=%d\n", a, b);
        return 0;
}
```

Durante a execução de ***Troca***, a memória estaria como mostrado.

Ou seja, seria como se no escopo da função, as variáveis ***a*** e ***b*** tivessem os apelidos de ***x*** e ***y***, que poderiam ser acessados nesse escopo.

Então as trocas ocorreriam também nos parâmetros reais!



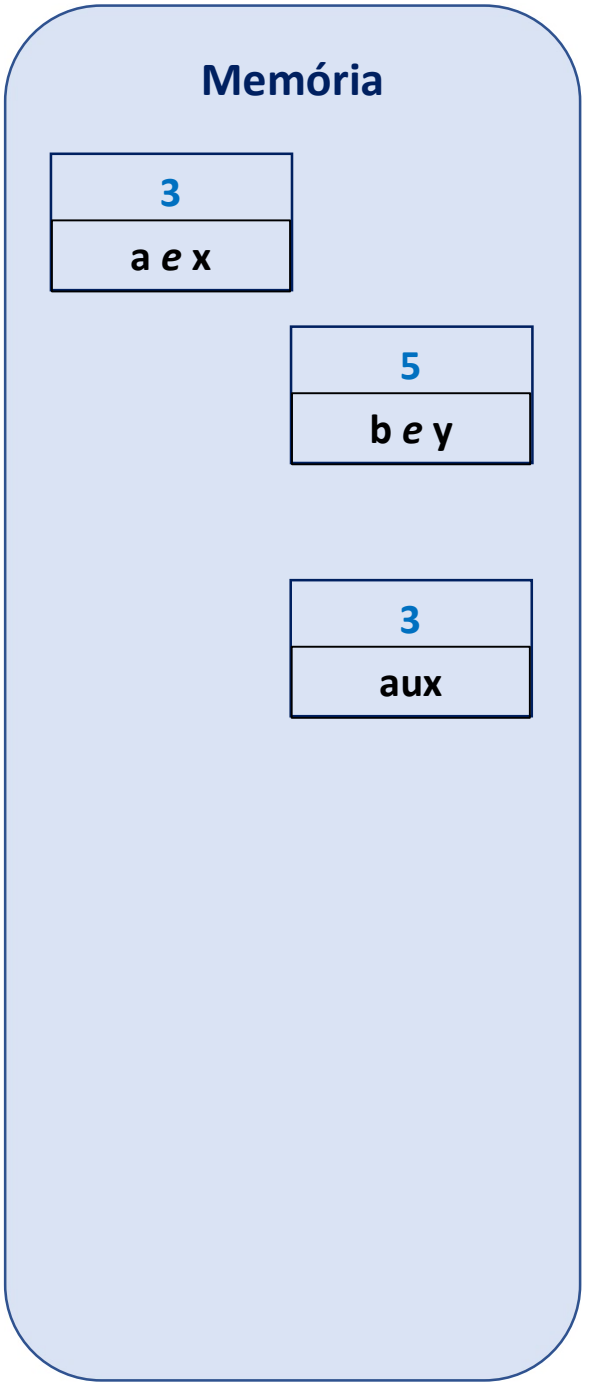
```c
#include <stdio.h>

void Troca(int x, int y){
        int aux = x;

        x = y;
        y = aux;
}

int main(void){
        int a = 3, b = 5;

        printf("antes: a=%d, b=%d\n", a, b);
        Troca(a, b);
        printf("depois: a=%d, b=%d\n", a, b);
        return 0;
}
```

Isso não existe em C!

Durante a execução de ***Troca***, a memória estaria como mostrado.

Ou seja, seria como se no escopo da função, as variáveis ***a*** e ***b*** tivessem os apelidos de ***x*** e ***y***, que poderiam ser acessados nesse escopo.

Então as trocas ocorreriam também nos parâmetros reais!



```c
#include <stdio.h>

void Troca(int x, int y){
        int aux = x;

        x = y;
        y = aux;
}

int main(void){
        int a = 3, b = 5;

        printf("antes: a=%d, b=%d\n", a, b);
        Troca(a, b);
        printf("depois: a=%d, b=%d\n", a, b);
        return 0;
}
```

Em C, podemos fazer uso dos **Ponteiros** para simular a passagem de parâmetros por referência.

Então os parâmetros formais podem fazer referência, **apontar**, para a área de memória dos parâmetros reais.

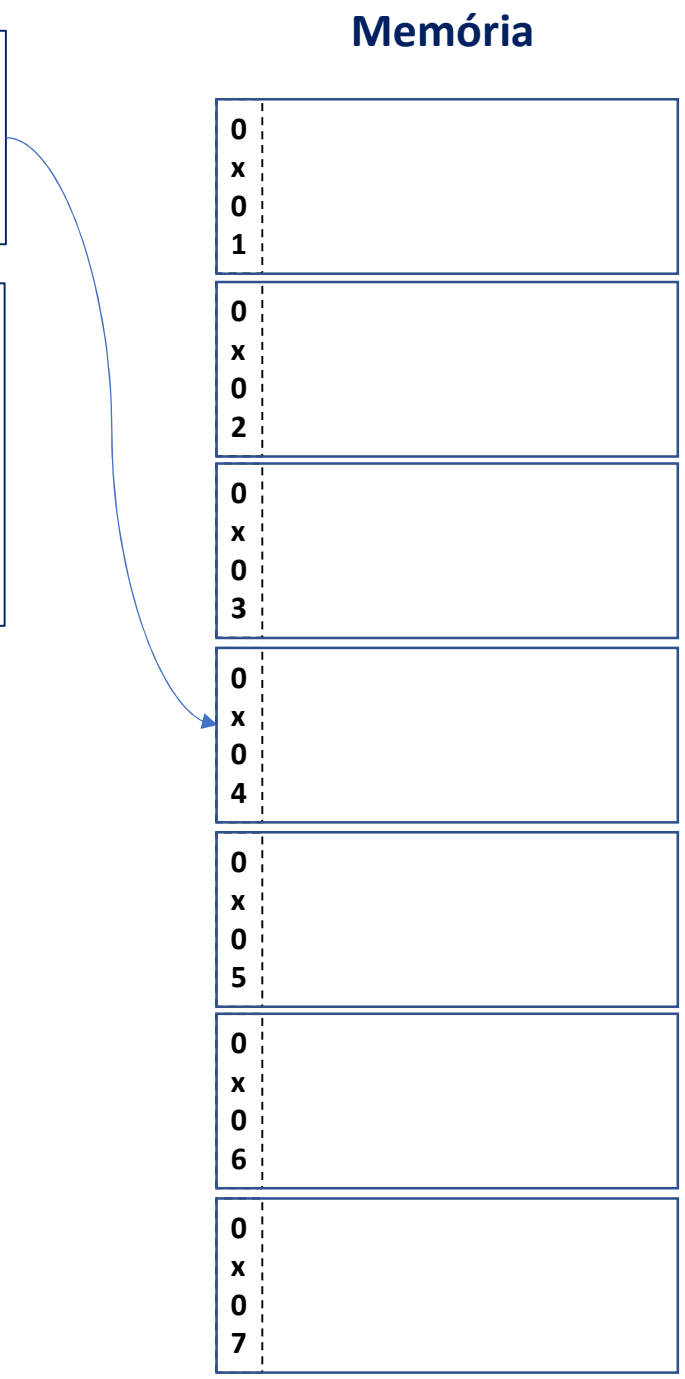
Ponteiros

Introdução à Programação

Profª. Giorgia Mattos – giorgiamattos@gmail.com

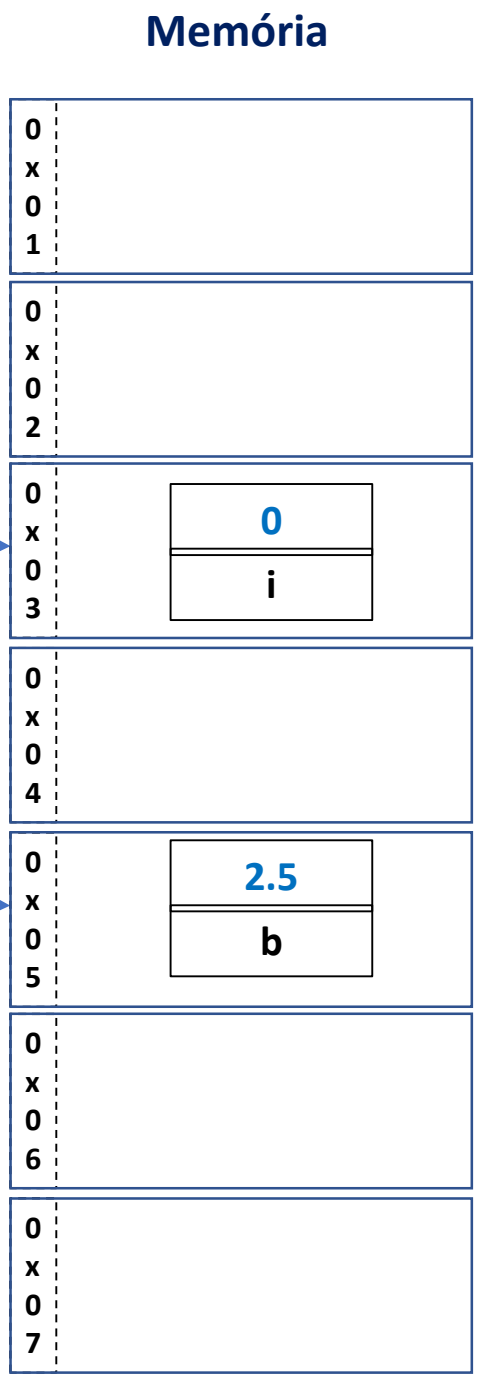
Vamos imaginar a memória como um espaço dividido em vários espaços menores.

Essa aqui tem sete espaços. Cada espaço possui seu próprio **endereço**. Aqui o endereço é representado por um número hexadecimal.



```
#include <stdio.h>

int main(void){

    int i = 0;
    float b = 2.5;

    return 0;
}
```

As variáveis são alocadas em espaços vazios da memória.

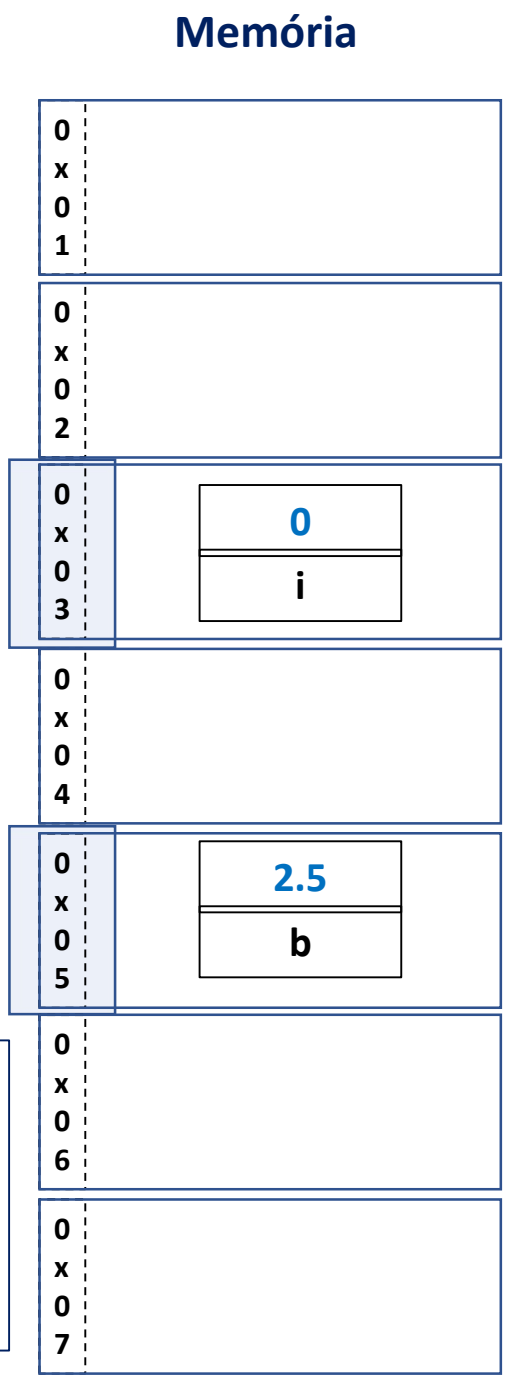


```c
#include <stdio.h>

int main(void){

    int i = 0;
    float b = 2.5;

    printf("end de i: %p\n", &i);
    printf("end de b: %p\n", &b);

    return 0;
}
```

O uso do operador **&** *com uma variável sempre* **resulta** no **endereço** dessa **variável.**

%p imprime endereços de memória (*ponteiros*).

Sendo assim, os dois ***printfs*** imprimem os endereços de memória onde as variáveis ***i*** e ***b*** estão alocadas, ou seja imprimem:
0x03
0x05



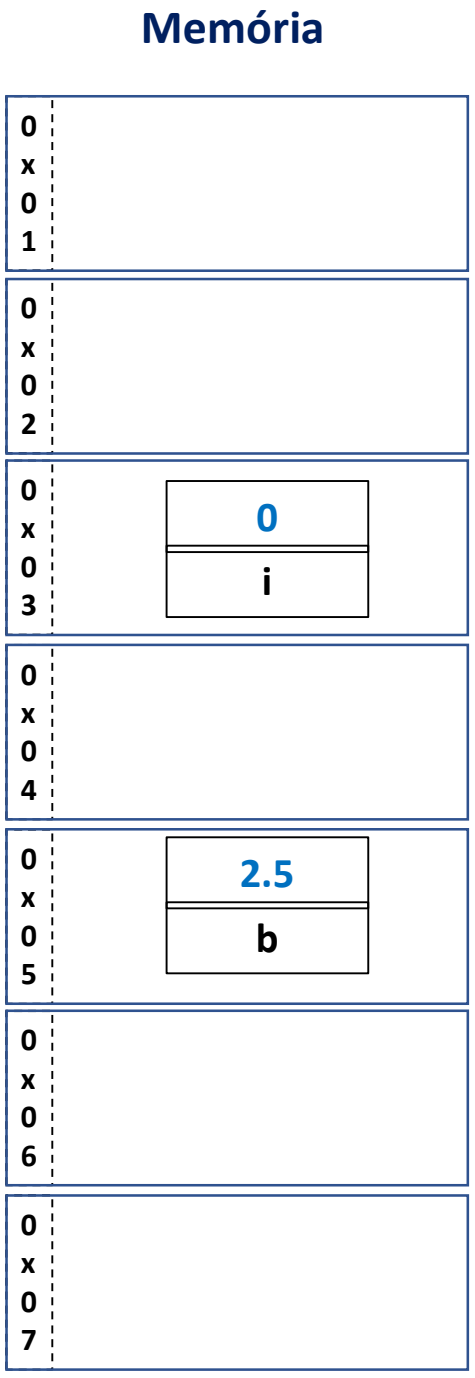
```
#include <stdio.h>

int main(void){

    int i = 0;
    float b = 2.5;
    int *p;

    return 0;
}
```

Ponteiros são variáveis como outras quaisquer.

Eles são sempre **ponteiros** de um ***tipo específico*** (qualquer tipo).

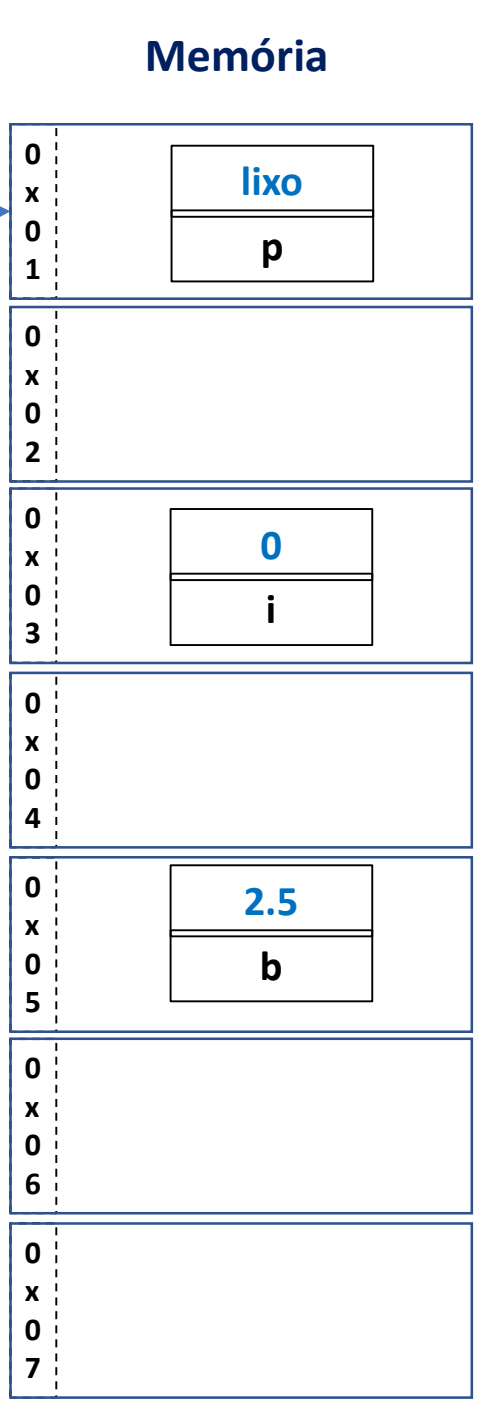
E apresentam, nas suas definições, esse ******* entre o tipo apontado e o nome da ***variável ponteiro.***

Então o nome da variável é ***p.***
p é um ponteiro para inteiros.



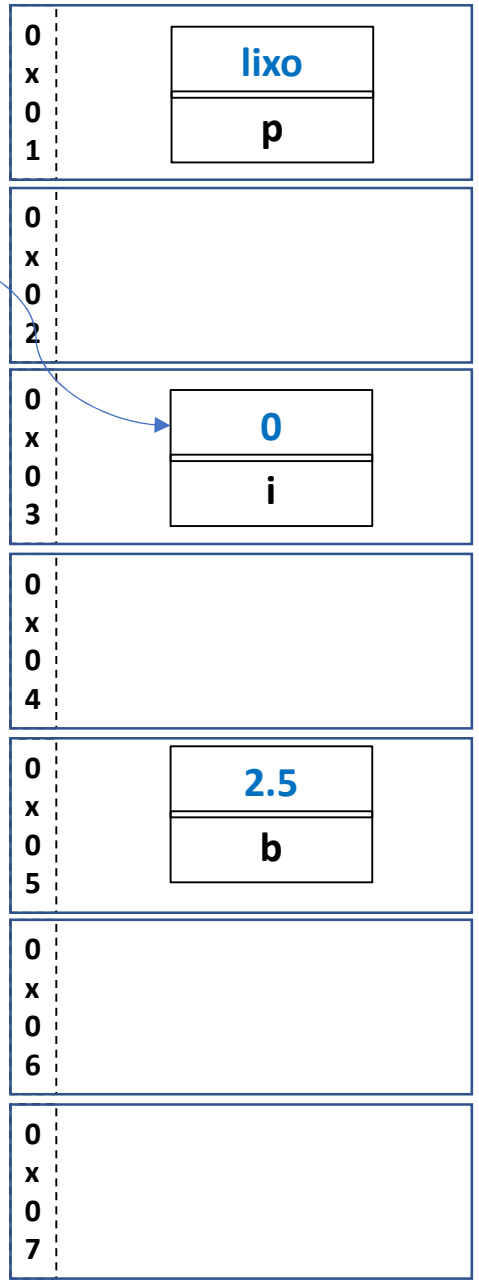
```c
#include <stdio.h>

int main(void){

    int i = 0;
    float b = 2.5;
    int *p;

    printf("end de i: %p\n", &i);
    printf("end de b: %p\n", &b);

    return 0;
}
```

Como as demais variáveis, ***p*** também deve ser alocado em um espaço vazio na memória.



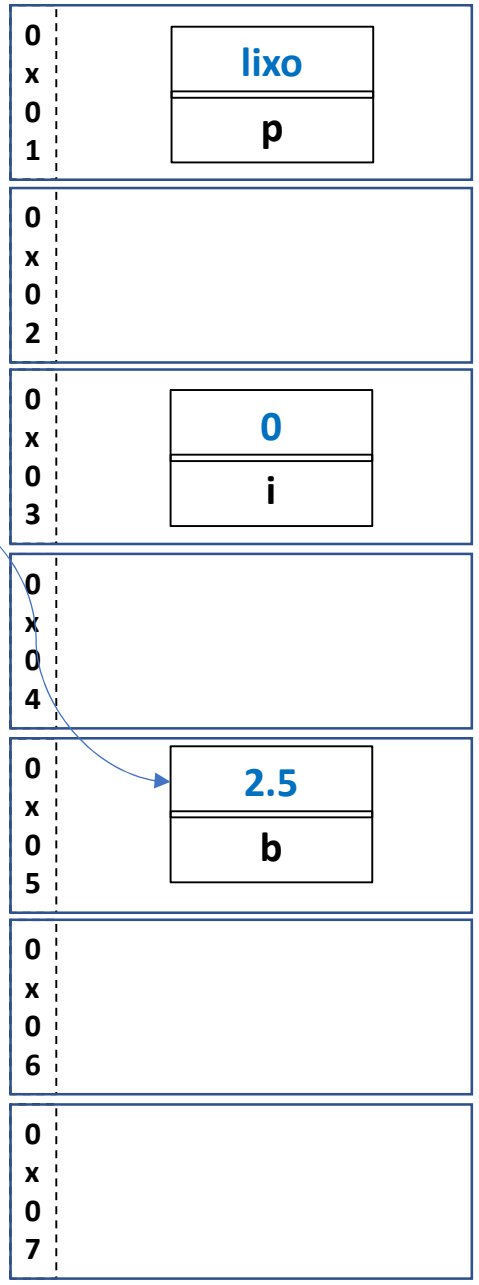
```c
#include <stdio.h>

int main(void){

        int i = 0;
        float b = 2.5;
        int *p;



        printf("end de i: %p\n", &i);
        printf("end de b: %p\n", &b);




        return 0;
}
```

Por ser do tipo ***int***, ***i*** armazena números inteiros.



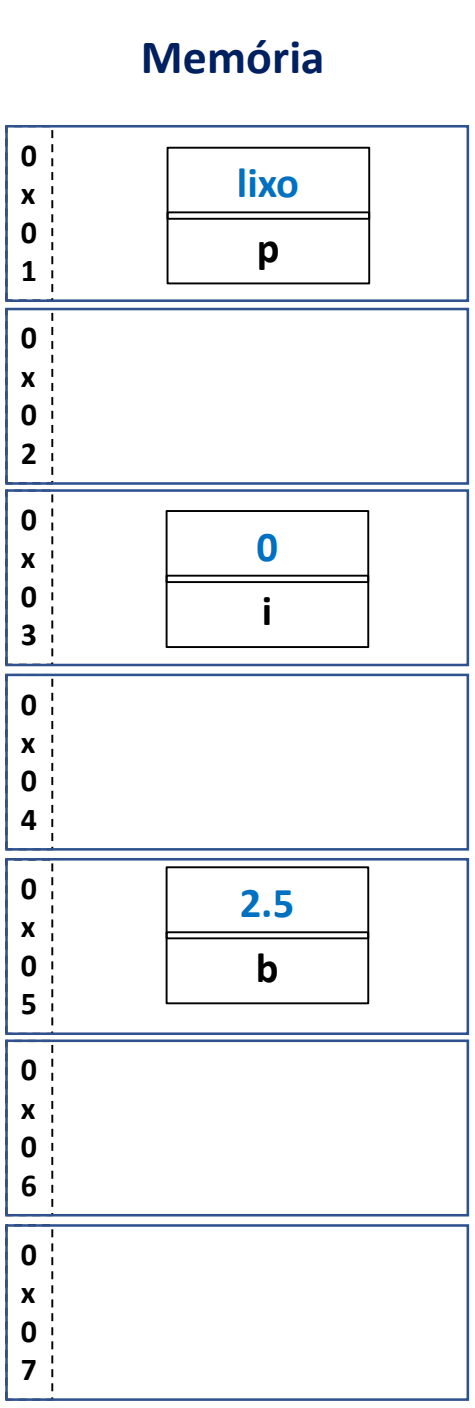
```c
#include <stdio.h>

int main(void){

        int i = 0;
        float b = 2.5;
        int *p;

        printf("end de i: %p\n", &i);
        printf("end de b: %p\n", &b);

        return 0;
}
```

Por ser do tipo ***float***, ***b*** armazena números reais.



```c
#include <stdio.h>

int main(void){

        int i = 0;
        float b = 2.5;
        int *p;

        printf("end de i: %p\n", &i);
        printf("end de b: %p\n", &b);



        return 0;
}
```

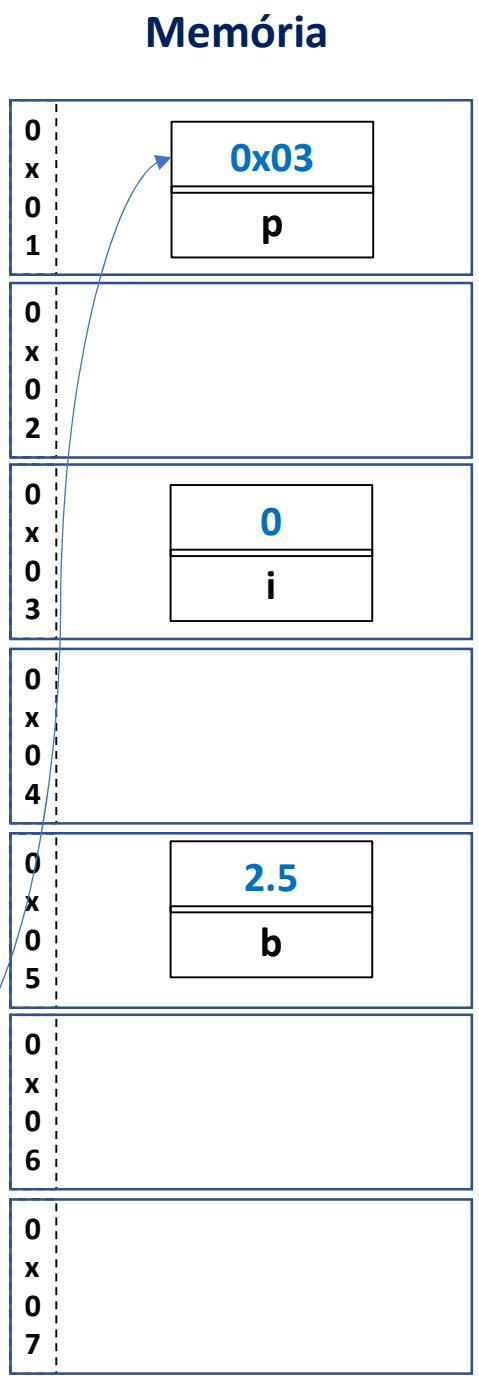
Por ser do tipo ***ponteiro para int***, ***p*** deve armazenar o endereço de memória de uma variável ***int***.



```c
#include <stdio.h>

int main(void){

    int i = 0;
    float b = 2.5;
    int *p;

    printf("end de i: %p\n", &i);
    printf("end de b: %p\n", &b);

    p = &i;

    return 0;
}
```

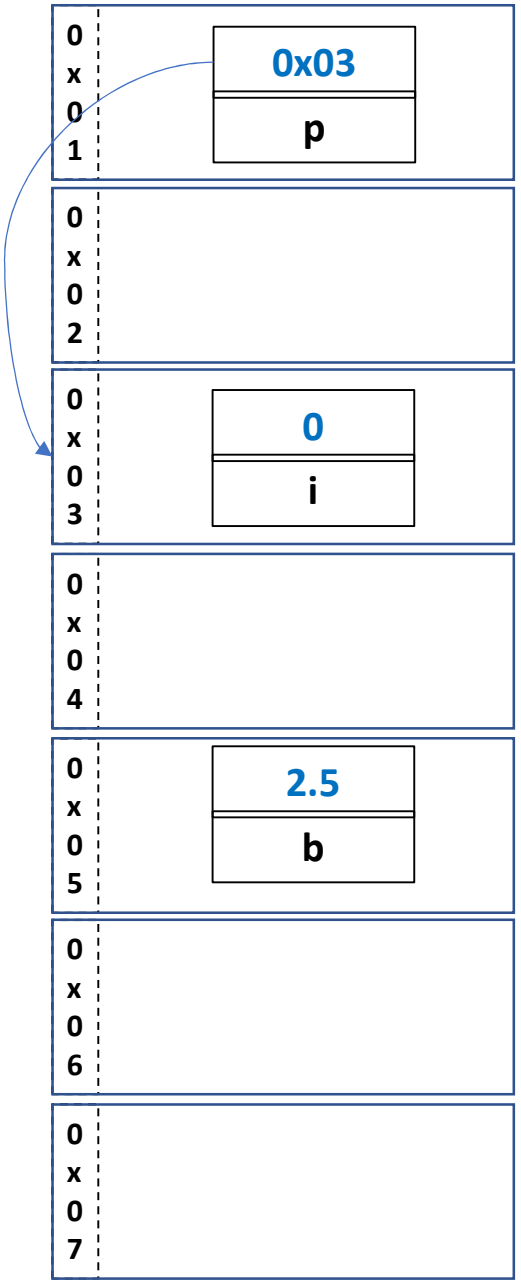
Por ser do tipo ***ponteiro para int***, ***p*** deve armazenar o endereço de memória de uma variável ***int***.

Essa atribuição faz com que ***p*** armazene o **endereço** da variável ***i***, que é ***0x03***.



```c
#include <stdio.h>

int main(void){

        int i = 0;
        float b = 2.5;
        int *p;




        printf("end de i: %p\n", &i);
        printf("end de b: %p\n", &b);

        p = &i;


        return 0;
}
```

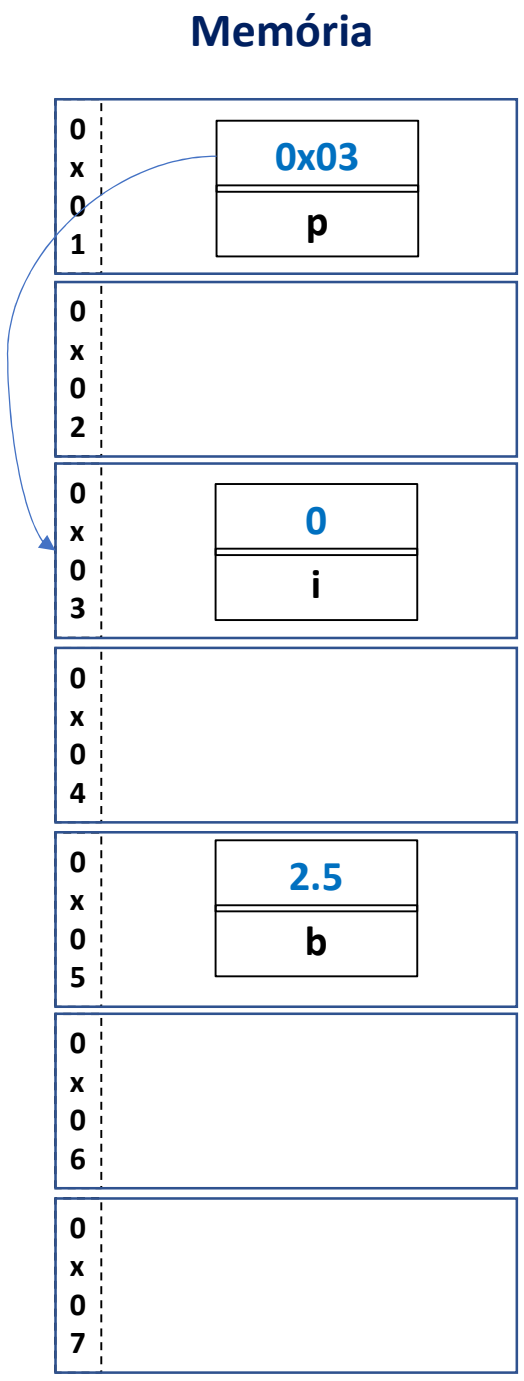
Podemos dizer que, agora, ***p aponta para i.***



```c
#include <stdio.h>

int main(void){

    int i = 0;
    float b = 2.5;
    int *p;

    printf("end de i: %p\n", &i);
    printf("end de b: %p\n", &b);

    p = &i;

    return 0;
}
```

Ou seja, deve ser possível acessar, ***indiretamente***, a variável alocada no endereço de memória armazenado em uma variável ponteiro.

Esse *acesso indireto* é possível com o uso do **operador de indireção: ***



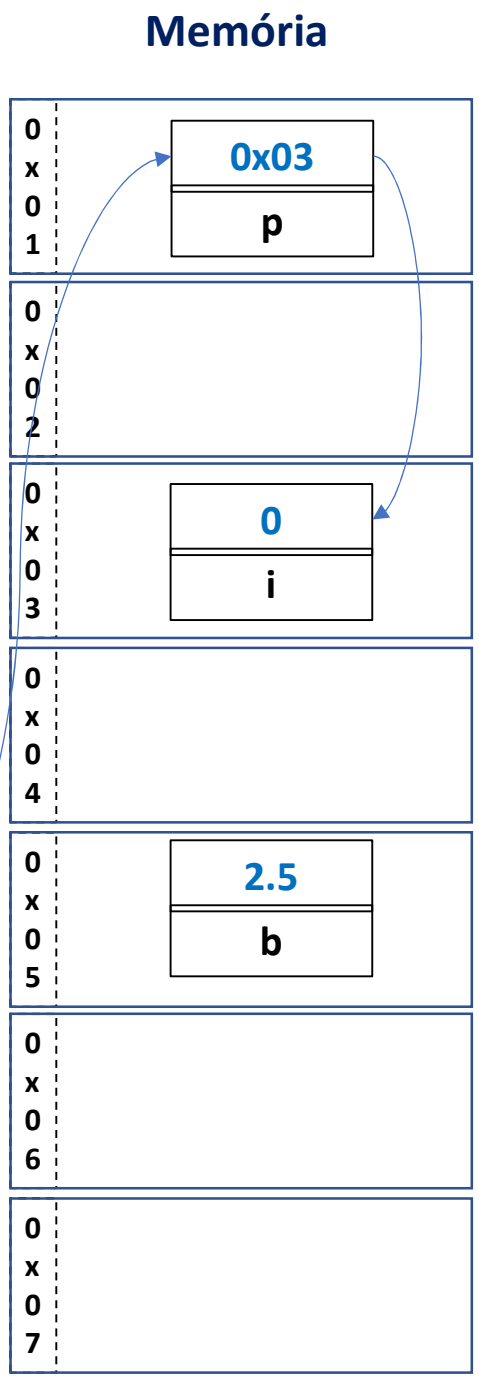
```c
#include <stdio.h>

int main(void){

        int i = 0;
        float b = 2.5;
        int *p;

        printf("end de i: %p\n", &i);
        printf("end de b: %p\n", &b);

        p = &i;
        printf("valor de *p: %d\n", *p);

        return 0;
}
```

Utilizado *sempre junto* a um **ponteiro**, o operador ***** tem como *resultado* o *valor* da *variável* alocada no *endereço armazenado* nesse **ponteiro.**

O ***printf*** imprime ***0.***

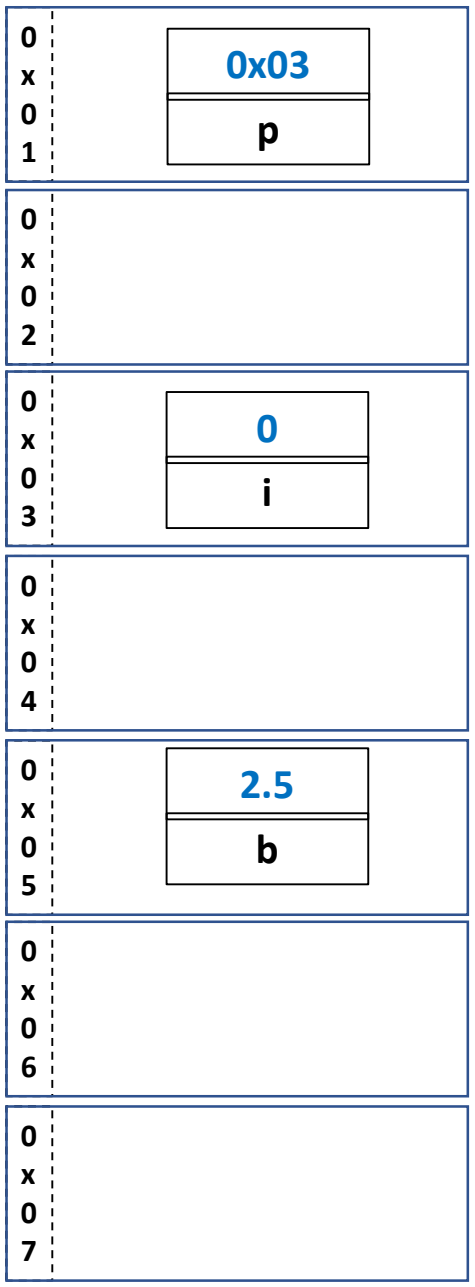
p armazena ***0x03.***

Então ****p*** é o valor da variável alocada em ***0x03.***



```c
#include <stdio.h>

int main(void){

    int i = 0;
    float b = 2.5;
    int *p;

    p = &i;
    printf("valor de *p: %d\n", *p);

    *p = 5;

    return 0;
}
```

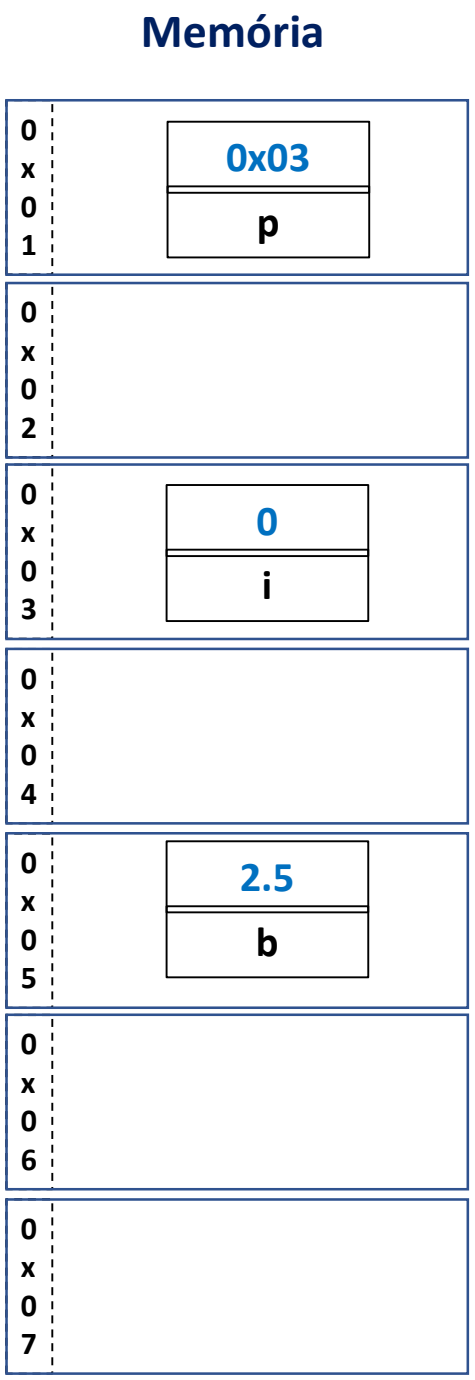
O operador * também pode ser usado em uma atribuição.



```c
#include <stdio.h>

int main(void){

        int i = 0;
        float b = 2.5;
        int *p;

        p = &i;
        printf("valor de *p: %d\n", *p);

        *p = 5;

        return 0;
}
```

O operador * também pode ser usado em uma atribuição.

Aqui, essa atribuição está armazenando o valor ***5*** na **variável** alocada no **endereço** armazenado em ***p***.

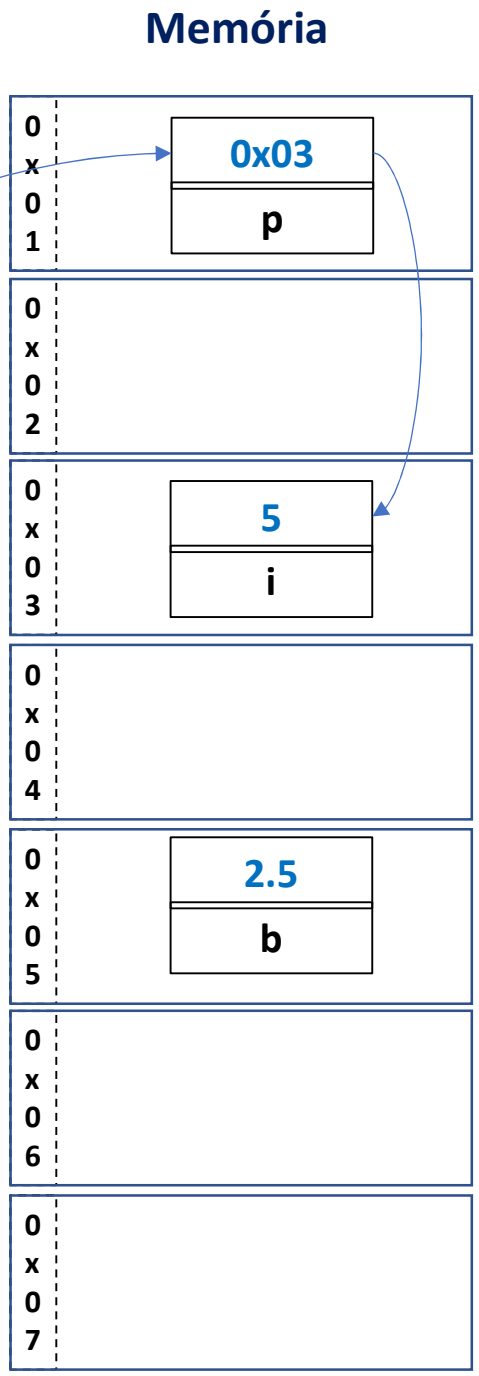


```c
#include <stdio.h>

int main(void){

        int i = 0;
        float b = 2.5;
        int *p;

        p = &i;
        printf("valor de *p: %d\n", *p);

        *p = 5;

        return 0;
}
```

O operador ***** também pode ser usado em uma atribuição.

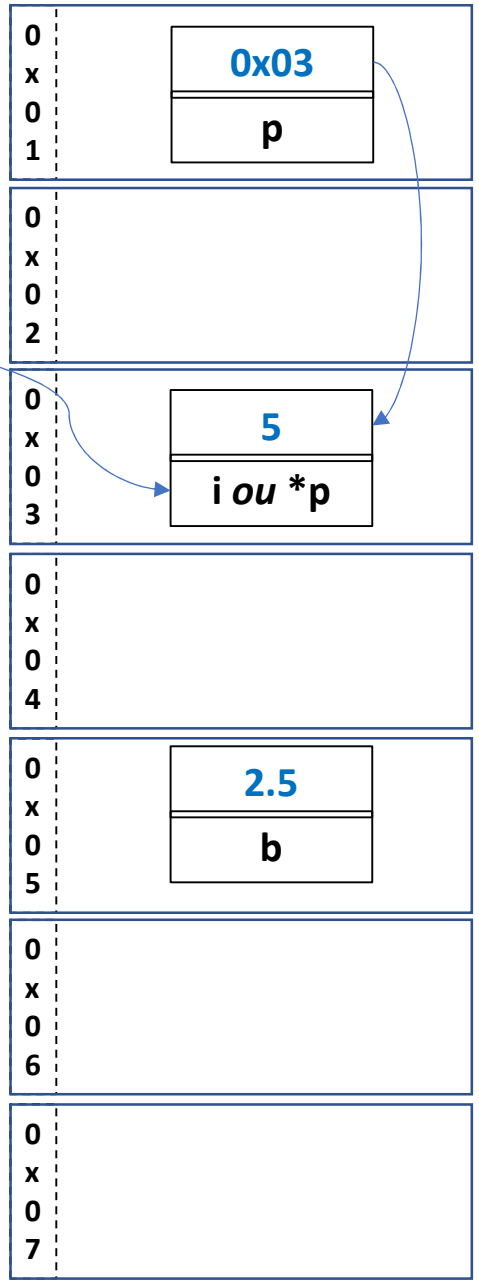
Aqui, essa atribuição está armazenando o valor ***5*** na **variável** alocada no **endereço** armazenado em ***p***.

Ou seja, está sendo **indiretamente atribuído** o valor **5** à variável ***i***.



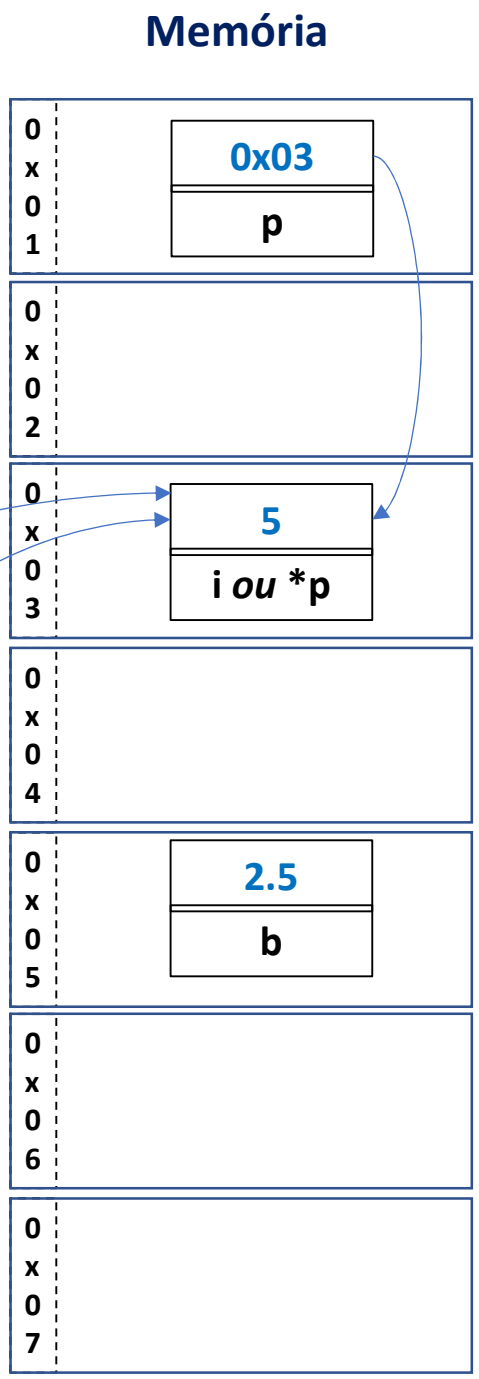
```c
#include <stdio.h>

int main(void){

        int i = 0;
        float b = 2.5;
        int *p;

        p = &i;
        printf("valor de *p: %d\n", *p);

        *p = 5;

        return 0;
}
```

Então, é como se, ao apontar para uma variável, o ponteiro criasse um *segundo nome* (ou *apelido*) para a variável apontada.



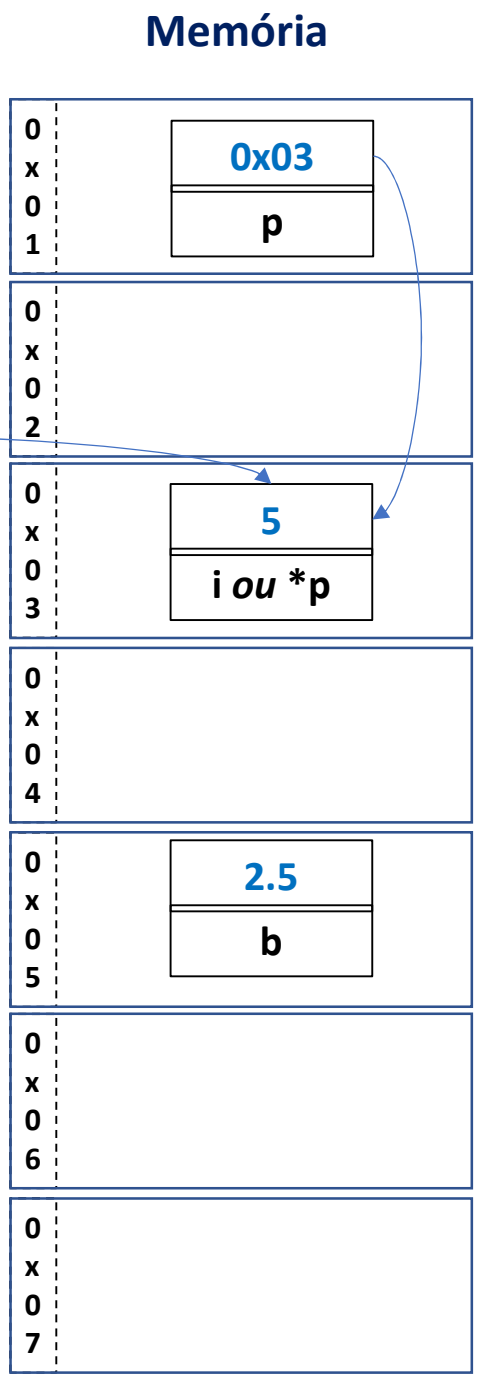
```c
#include <stdio.h>

int main(void){

    int i = 0;
    float b = 2.5;
    int *p;


    p = &i;
    printf("valor de *p: %d\n", *p);

    *p = 5;

    return 0;
}
```

Sendo assim, *em qualquer situação*, o uso do ***operador*** ***** em um ponteiro faz referência ***indireta*** a região de memória da variável apontada. Ou seja, ao usar o ***** antes de um ponteiro, estamos referenciando a variável que está alocada no endereço de memória armazenado no ponteiro.



```c
#include <stdio.h>

int main(void){

        int i = 0;
        float b = 2.5;
        int *p;

        p = &i;
        printf("valor de *p: %d\n", *p);
        *p = 5;
         printf("valor de i: %d\n", i);
        return 0;
}
```

Então esse ***printf*** irá imprimir ***5*** para o valor de ***i***, já que ele foi *indiretamente* alterado na instrução anterior.



```c
#include <stdio.h>

void Troca(int x, int y){
        int aux = x;

        x = y;
        y = aux;
}

int main(void){
        int a = 3, b = 5;

        printf("antes: a=%d, b=%d\n", a, b);
        Troca(a, b);
        printf("depois: a=%d, b=%d\n", a, b);
        return 0;
}
```

Em C, podemos fazer uso dos **Ponteiros** para simular a passagem de parâmetros por referência.

Então os parâmetros formais podem fazer referência, **apontar**, para a área de memória dos parâmetros reais.

Vamos atualizar o código-fonte para fazer o uso de **ponteiros** e simular a passagem por referência, alterando os valores de ***a*** e ***b*** através da função ***Troca***.

Nova versão.

```c
#include <stdio.h>

void Troca(int *x, int *y){
        int aux = *x;

        *x = *y;
        *y = aux;
}



int main(void){
        int a = 3, b = 5;

        printf("antes: a=%d, b=%d\n", a, b);
        Troca(&a, &b);
        printf("depois: a=%d, b=%d\n", a, b);
        return 0;
}
```

Nova versão.

```c
#include <stdio.h>

void Troca(int *x, int *y){
        int aux = *x;

        *x = *y;
        *y = aux;
}

int main(void){
        int a = 3, b = 5;

        printf("antes: a=%d, b=%d\n", a, b);
        Troca(&a, &b);
        printf("depois: a=%d, b=%d\n", a, b);
        return 0;
}
```

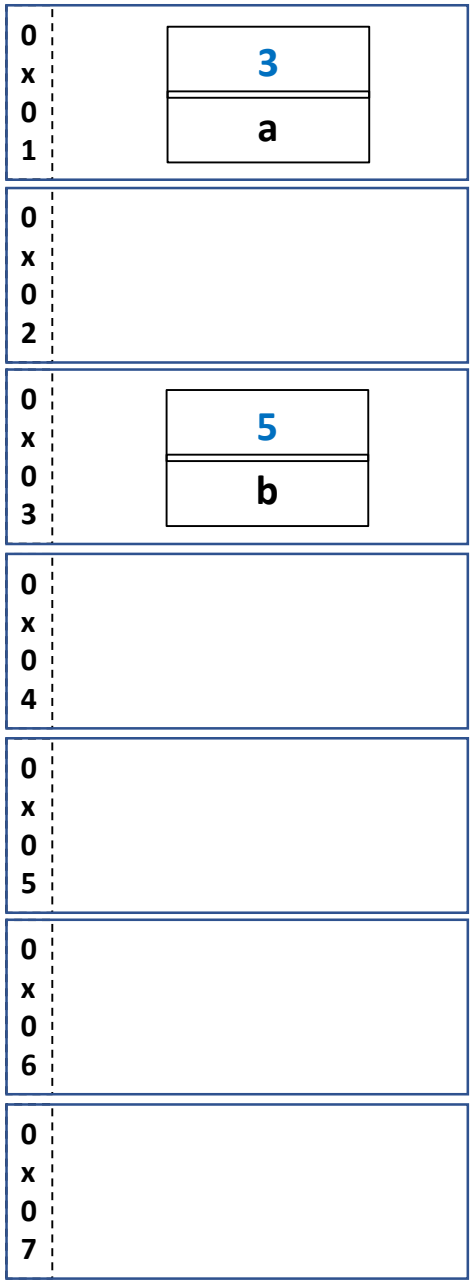
Os parâmetros agora são ponteiros.

Analisemos os efeitos desta e das outras modificações no código-fonte enquanto acompanhamos seu funcionamento...

```c
#include <stdio.h>

void Troca(int *x, int *y){
        int aux = *x;

        *x = *y;
        *y = aux;
}

int main(void){
        int a = 3, b = 5;

        printf("antes: a=%d, b=%d\n", a, b);
        Troca(&a, &b);
        printf("depois: a=%d, b=%d\n", a, b);
        return 0;
}
```

Como já vimos, ao chegar no primeiro ***printf***, a memória do computador apresentará as variáveis **a** e **b** alocadas com seus valores.

Memória

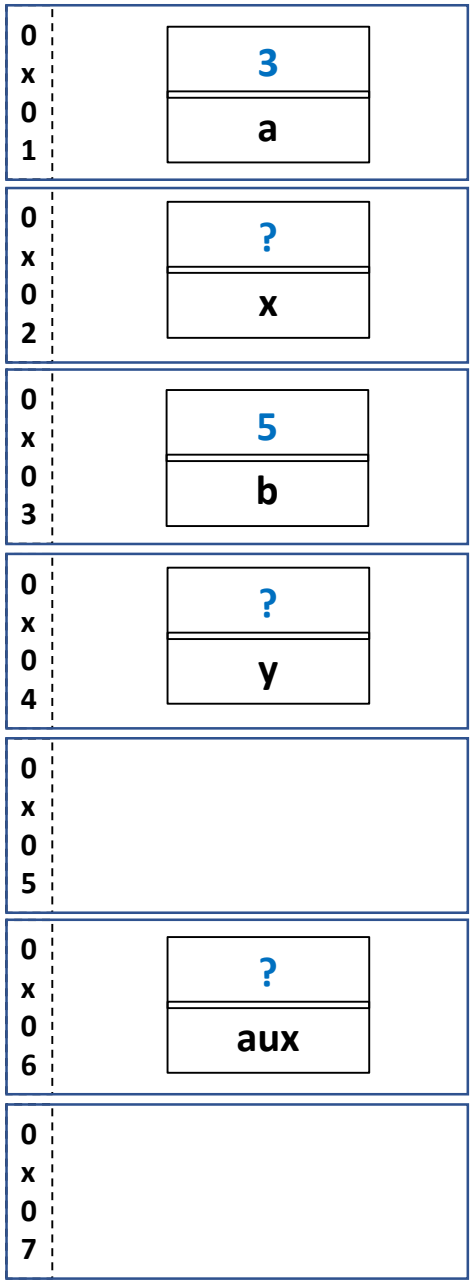


```c
#include <stdio.h>

void Troca(int *x, int *y){
        int aux = *x;

        *x = *y;
        *y = aux;
}

int main(void){
        int a = 3, b = 5;

        printf("antes: a=%d, b=%d\n", a, b);
        Troca(&a, &b);
        printf("depois: a=%d, b=%d\n", a, b);
        return 0;
}
```

A preparação para execução da função ***Troca*** alocará os parâmetros e as variáveis de duração automática do bloco da função.

Sabemos que os parâmetros formais recebem uma cópia dos valores dos parâmetros reais.

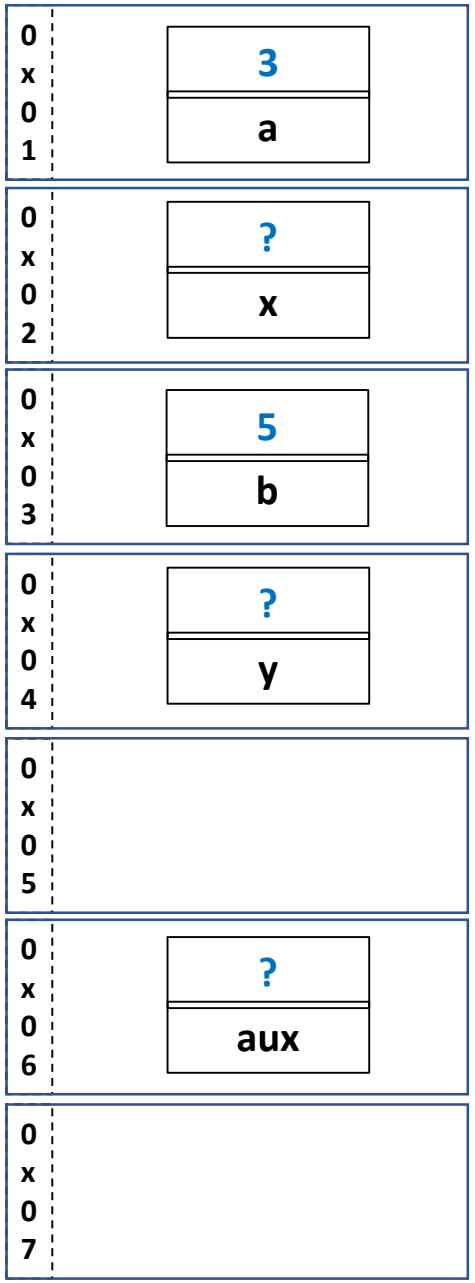
Memória



```c
#include <stdio.h>

void Troca(int *x, int *y){
        int aux = *x;

        *x = *y;
        *y = aux;
}

int main(void){
        int a = 3, b = 5;

        printf("antes: a=%d, b=%d\n", a, b);
        Troca(&a, &b);
        printf("depois: a=%d, b=%d\n", a, b);
        return 0;
}
```

Mas os parâmetros reais agora são os endereços das variáveis ***a*** e ***b***.

x e ***y*** recebem uma cópia desses valores.

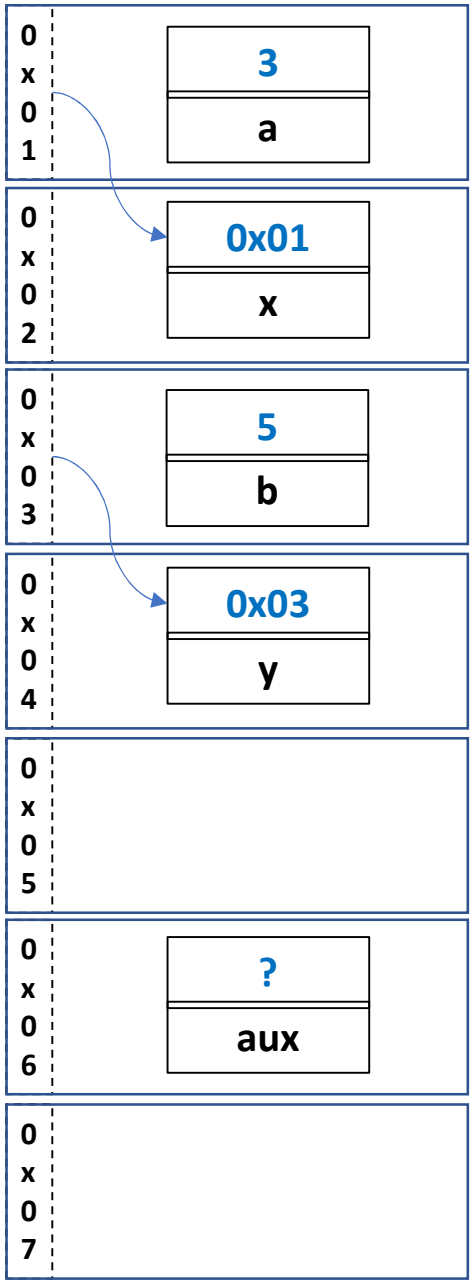


```c
#include <stdio.h>

void Troca(int *x, int *y){
        int aux = *x;

        *x = *y;
        *y = aux;
}

int main(void){
        int a = 3, b = 5;

        printf("antes: a=%d, b=%d\n", a, b);
        Troca(&a, &b);
        printf("depois: a=%d, b=%d\n", a, b);
        return 0;
}
```

Mas os parâmetros reais agora são os endereços das variáveis ***a*** e ***b***.

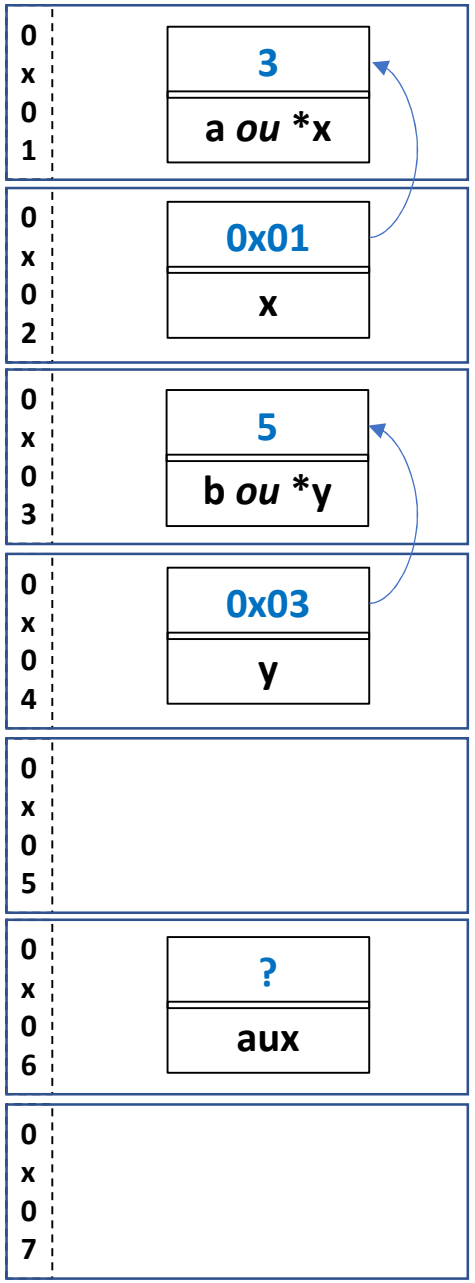
x e ***y*** recebem uma cópia desses valores.

Memória



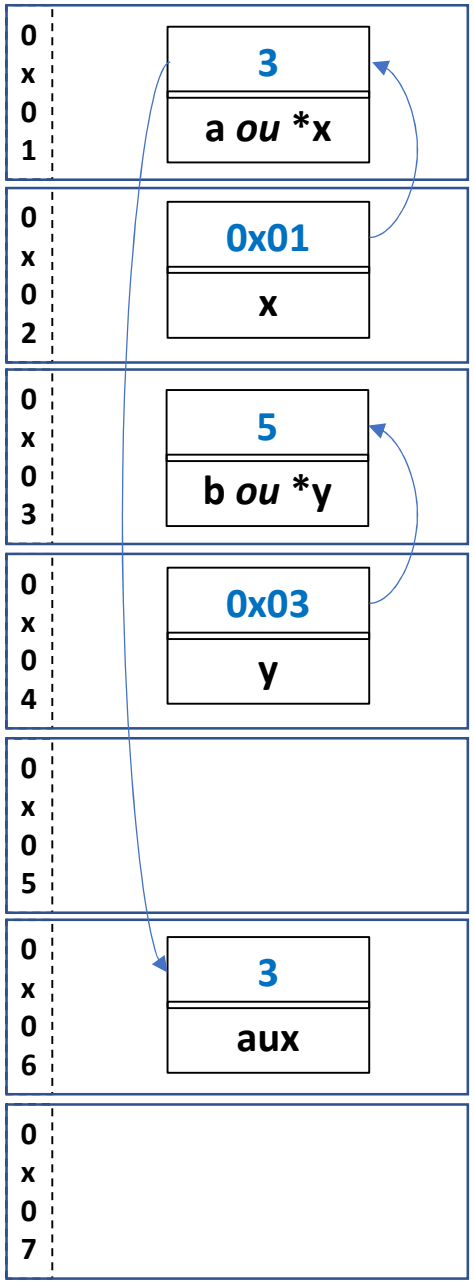
```c
#include <stdio.h>

void Troca(int *x, int *y){
        int aux = *x;

        *x = *y;
        *y = aux;
}

int main(void){
        int a = 3, b = 5;

        printf("antes: a=%d, b=%d\n", a, b);
        Troca(&a, &b);
        printf("depois: a=%d, b=%d\n", a, b);
        return 0;
}
```

Cria-se então a **ligação** entre as variáveis.



```c
#include <stdio.h>

void Troca(int *x, int *y){
        int aux = *x;

        *x = *y;
        *y = aux;
}

int main(void){
        int a = 3, b = 5;

        printf("antes: a=%d, b=%d\n", a, b);
        Troca(&a, &b);
        printf("depois: a=%d, b=%d\n", a, b);
        return 0;
}
```

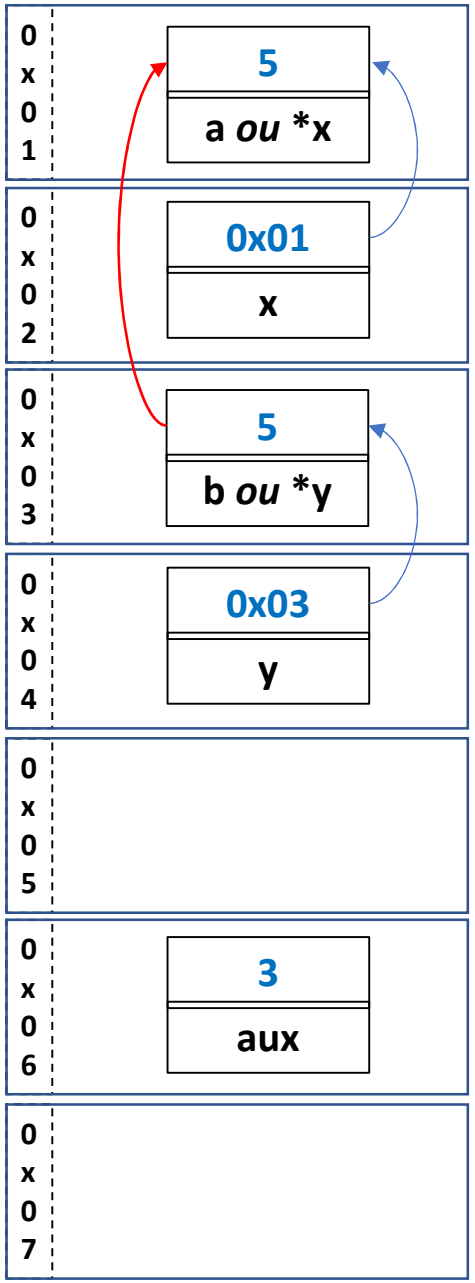
E a inicialização de ***aux*** faz ela receber, via *acesso indireto* do **operador *,** o valor da variável apontada por ***x***.



```c
#include <stdio.h>

void Troca(int *x, int *y){
        int aux = *x;

        *x = *y;
        *y = aux;
}

int main(void){
        int a = 3, b = 5;

        printf("antes: a=%d, b=%d\n", a, b);
        Troca(&a, &b);
        printf("depois: a=%d, b=%d\n", a, b);
        return 0;
}
```

Através do *acesso indireto* nos ponteiros ***x*** e ***y***, a variável apontada por ***x*** (***a***) receberá o valor da variável apontada por ***y*** (***b***).

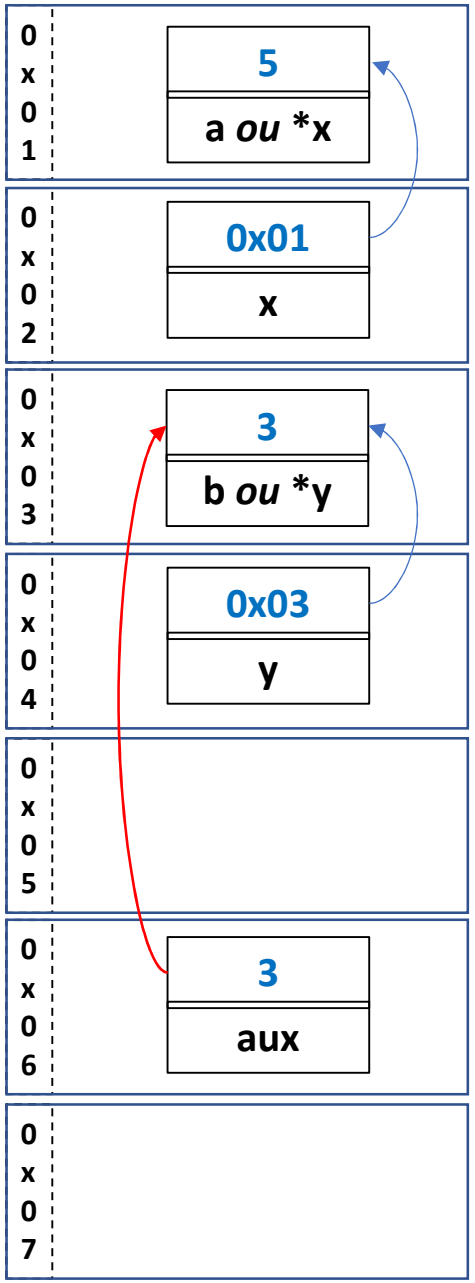
Memória



```c
#include <stdio.h>

void Troca(int *x, int *y){
        int aux = *x;

        *x = *y;
        *y = aux;
}

int main(void){
        int a = 3, b = 5;

        printf("antes: a=%d, b=%d\n", a, b);
        Troca(&a, &b);
        printf("depois: a=%d, b=%d\n", a, b);
        return 0;
}
```

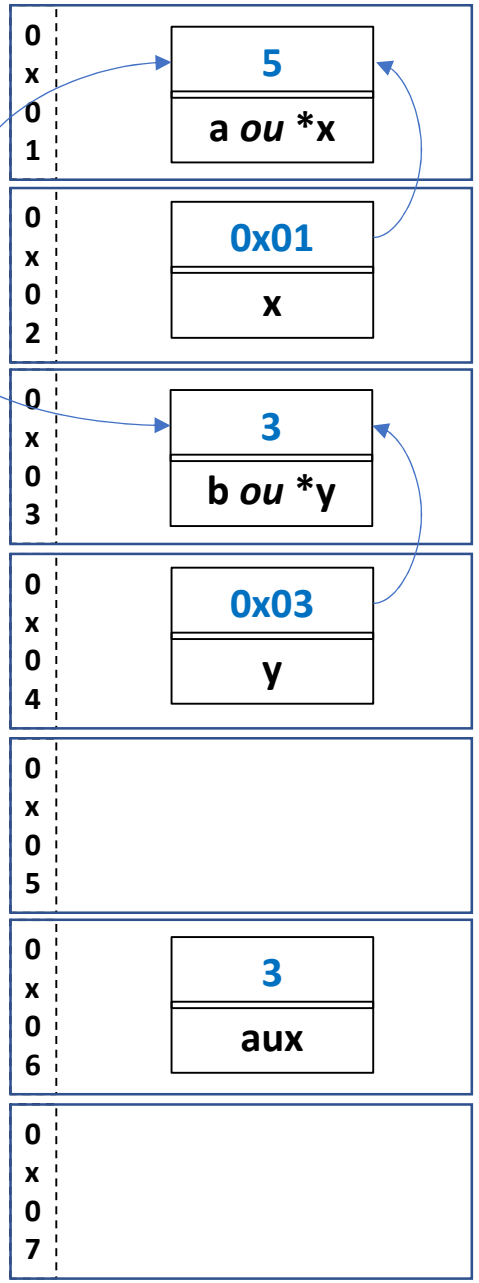
Através do *acesso indireto* no ponteiro ***y***, a variável apontada por ***y*** (***b***) receberá o valor da variável ***aux***.

Memória



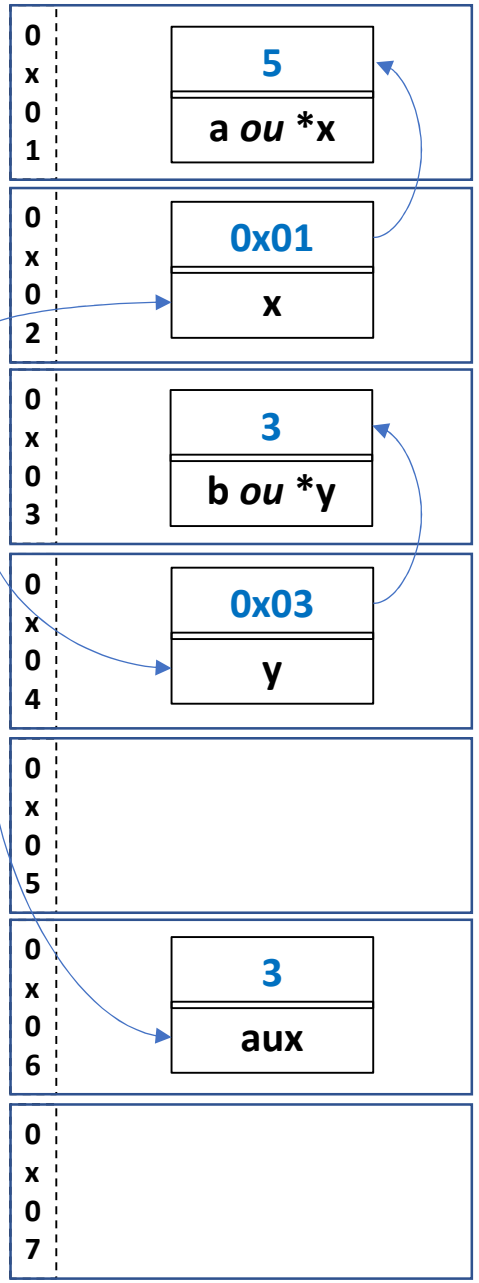
```c
#include <stdio.h>

void Troca(int *x, int *y){
        int aux = *x;

        *x = *y;
        *y = aux;
}

int main(void){
        int a = 3, b = 5;

        printf("antes: a=%d, b=%d\n", a, b);
        Troca(&a, &b);
        printf("depois: a=%d, b=%d\n", a, b);
        return 0;
}
```

Devido ao acesso indireto através dos ponteiros, as variáveis ***a*** e ***b*** tiveram seus valores alterados.



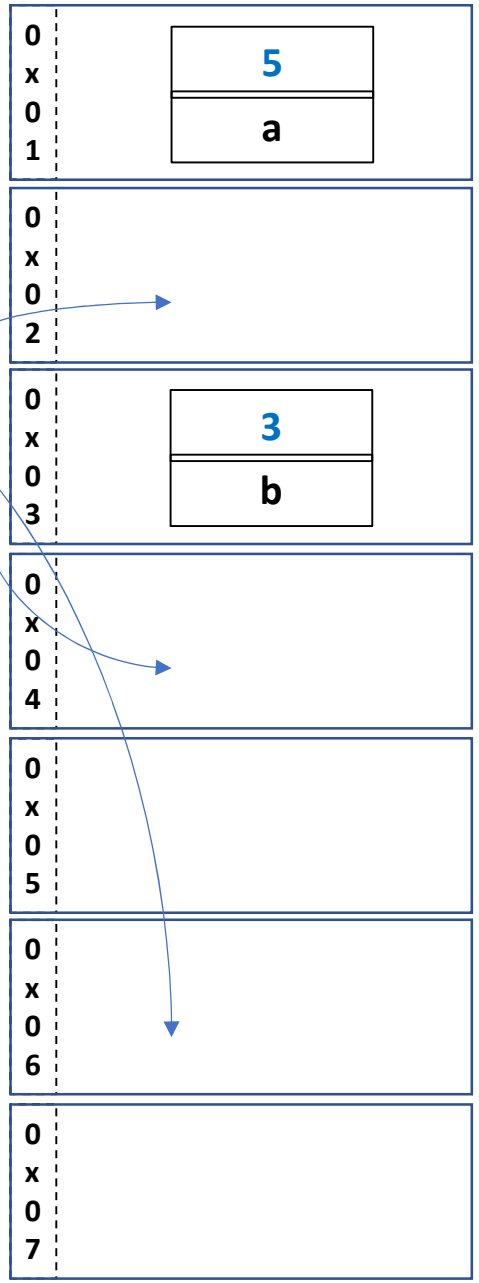
```c
#include <stdio.h>

void Troca(int *x, int *y){
        int aux = *x;

        *x = *y;
        *y = aux;
}

int main(void){
        int a = 3, b = 5;

        printf("antes: a=%d, b=%d\n", a, b);
        Troca(&a, &b);
        printf("depois: a=%d, b=%d\n", a, b);
        return 0;
}
```

Ao final da execução de ***Troca***, a variável de duração automática e os parâmetros serão liberados.



```c
#include <stdio.h>

void Troca(int *x, int *y){
        int aux = *x;

        *x = *y;
        *y = aux;
}

int main(void){
        int a = 3, b = 5;

        printf("antes: a=%d, b=%d\n", a, b);
        Troca(&a, &b);
        printf("depois: a=%d, b=%d\n", a, b);
        return 0;
}
```

Ao final da execução de ***Troca***, a variável de duração automática e os parâmetros serão liberados.

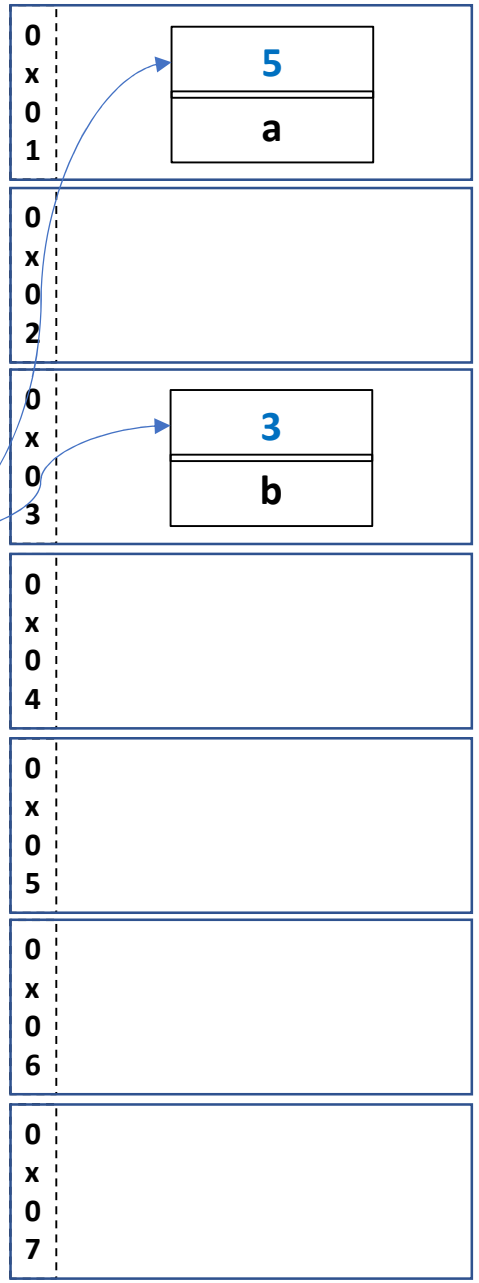


```c
#include <stdio.h>

void Troca(int *x, int *y){
        int aux = *x;

        *x = *y;
        *y = aux;
}

int main(void){
        int a = 3, b = 5;

        printf("antes: a=%d, b=%d\n", a, b);
        Troca(&a, &b);
        printf("depois: a=%d, b=%d\n", a, b);
        return 0;
}
```

Restando na memória as variáveis ***a*** e ***b***, com as alterações feitas.

O segundo ***printf*** imprime os novos valores de ***a*** e ***b***, efetivamente alterados pela função ***Troca*** através do uso de **ponteiros**.

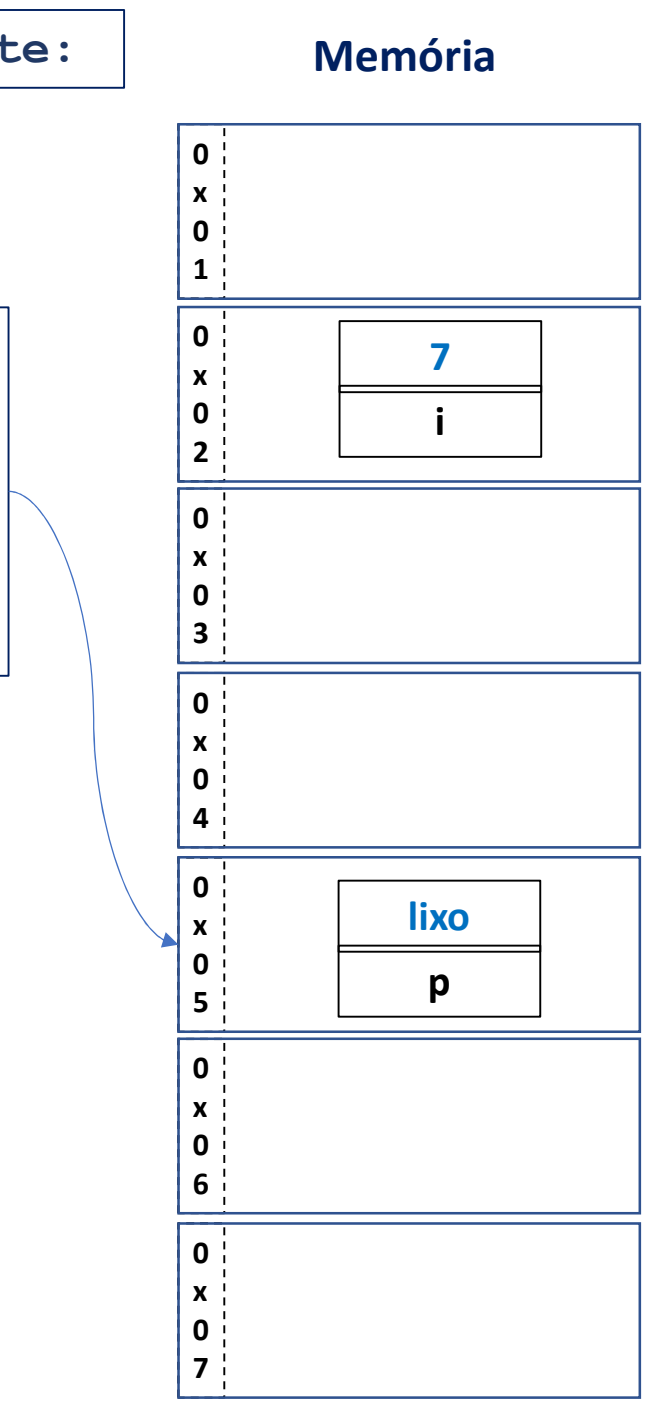
Memória



Observação importante:

```
#include <stdio.h>

int main(void){
        int i = 7;
        int *p;

        return 0;
}
```

Como as demais variáveis, ***p*** também é alocado em um espaço vazio na memória e, se não iniciado, receberá um lixo de memória.

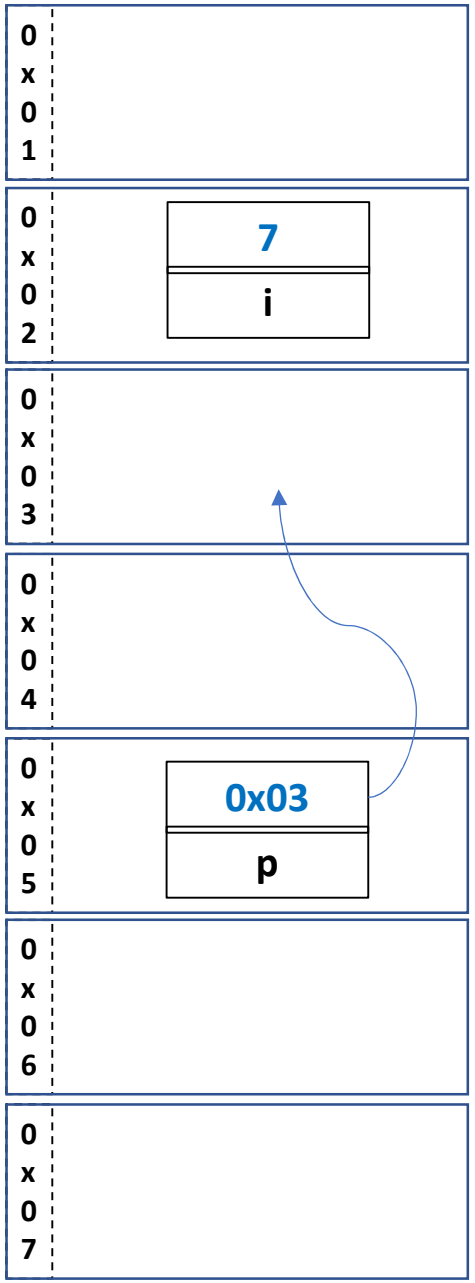


Observação importante:

```
#include <stdio.h>

int main(void){
        int i = 7;
        int *p;

        return 0;
}
```

Como as demais variáveis, ***p*** também é alocado em um espaço vazio na memória e, se não iniciado, receberá um lixo de memória.

Esse lixo será interpretado como um endereço de memória e pode indicar um endereço que não pertence a uma variável do programa.

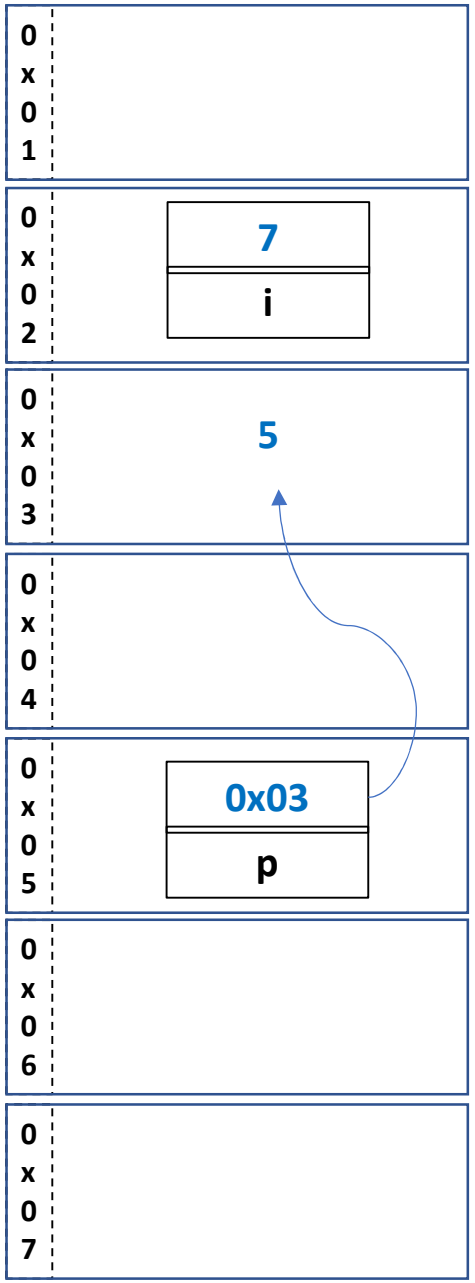


Observação importante:

```
#include <stdio.h>

int main(void){
        int i = 7;
        int *p;

        *p = 5;

        return 0;
}
```

Ao fazer uma indireção nesse ponteiro, em um endereço que não conhecemos, estaremos alterando uma região de memória que provavelmente não deveríamos. Essa região pode ser até uma variável de um outro programa.

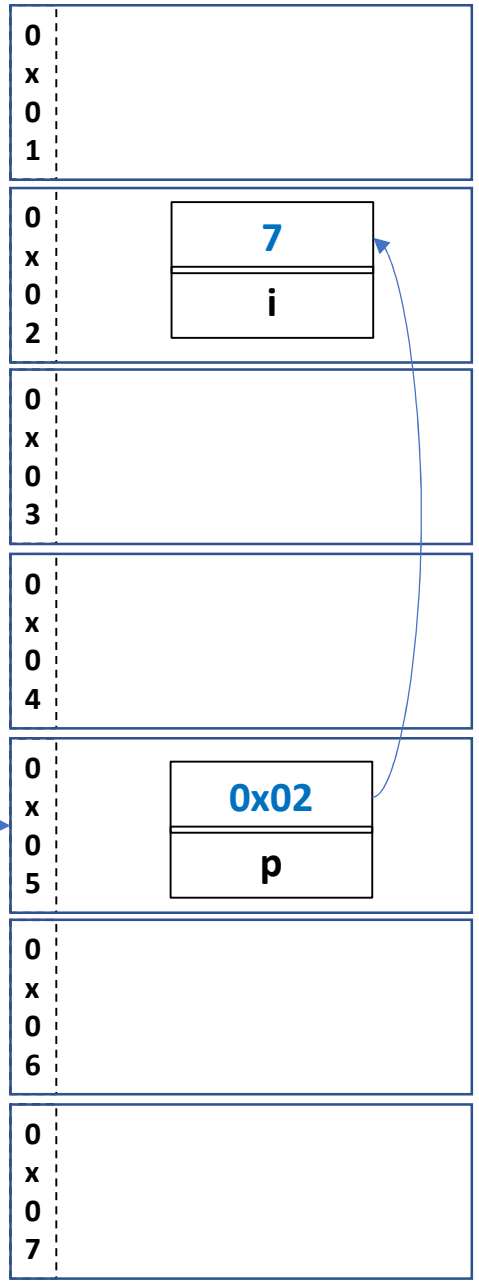
Na verdade o sistema operacional vai identificar esse acesso indevido e vai encerrar a execução do programa.
No entanto...



Observação importante:

```
#include <stdio.h>

int main(void){
        int i = 7;
        int *p;




        return 0;
}
```

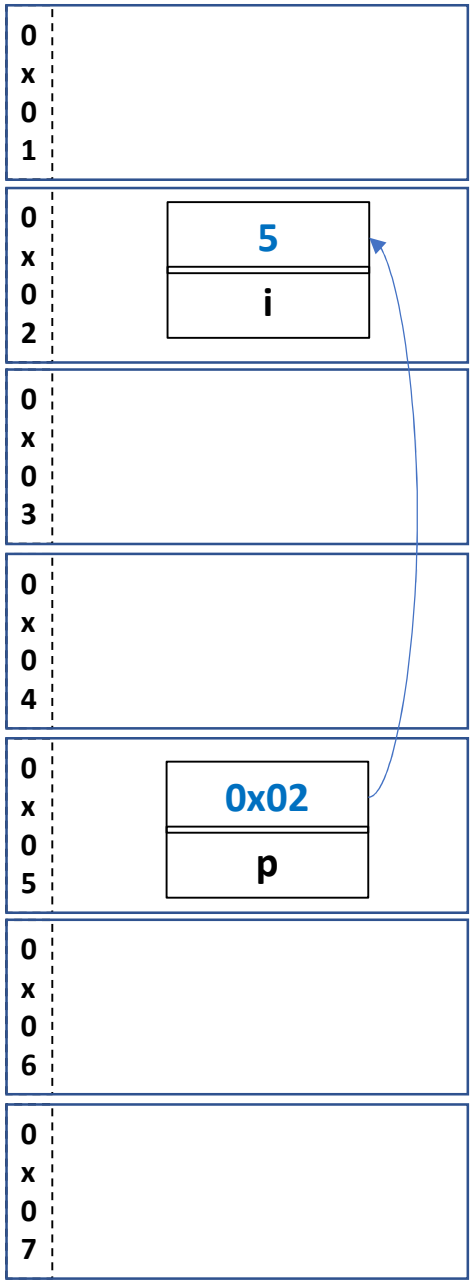
Por uma coincidência, o lixo de memória poderia indicar o endereço de uma variável do programa.



Observação importante:

```
#include <stdio.h>

int main(void){
        int i = 7;
        int *p;

        *p = 5;


        return 0;
}
```

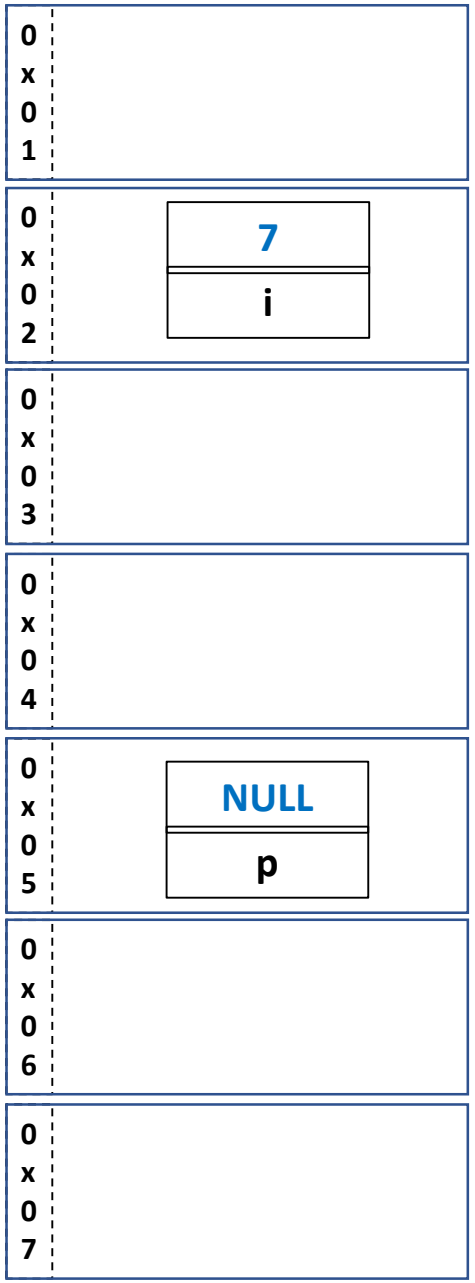
Agora, esse acesso indireto vai alterar a variável ***i*** e isso provavelmente não era a intenção, já que ***p*** não foi explicitamente iniciado com o endereço de ***i***.



Observação importante:

```
#include <stdio.h>

int main(void){
        int i = 7;
        int *p = NULL;

        return 0;
}
```

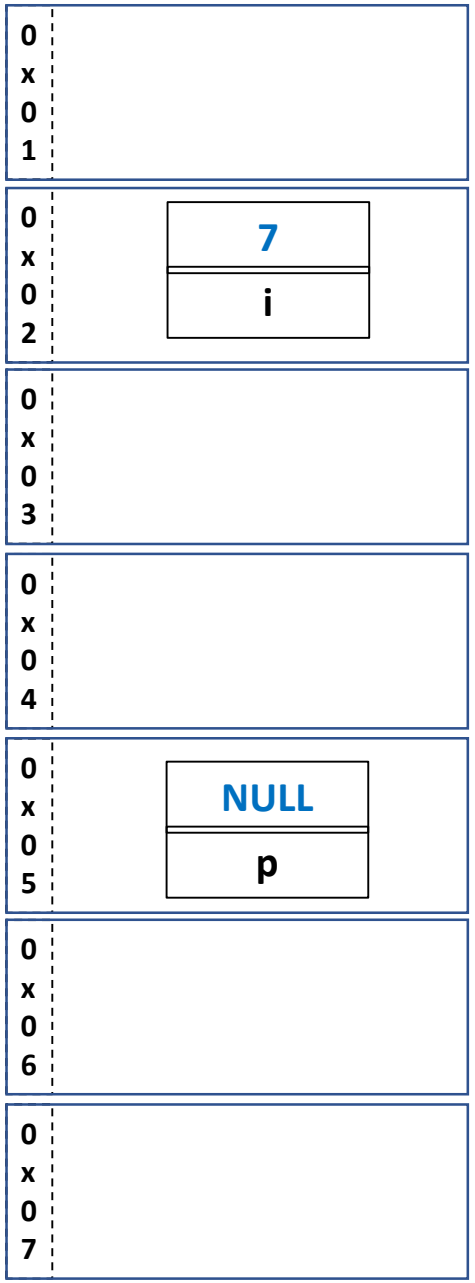
Portanto, é uma boa prática iniciar os ponteiros com a constante simbólica ***NULL***. Essa constante normalmente tem o valor ***0*** e indica que o ponteiro é nulo, ou seja, não aponta para local algum.



Observação importante:

```
#include <stdio.h>

int main(void){
        int i = 7;
        int *p = NULL;

        *p = 5;

        return 0;
}
```

O acesso indireto em um ponteiro nulo continuará encerrando o programa por falha de segmentação, mas pelo menos temos a certeza que nunca será alterado o valor de uma variável que por acaso teve seu endereço iniciado no ponteiro.

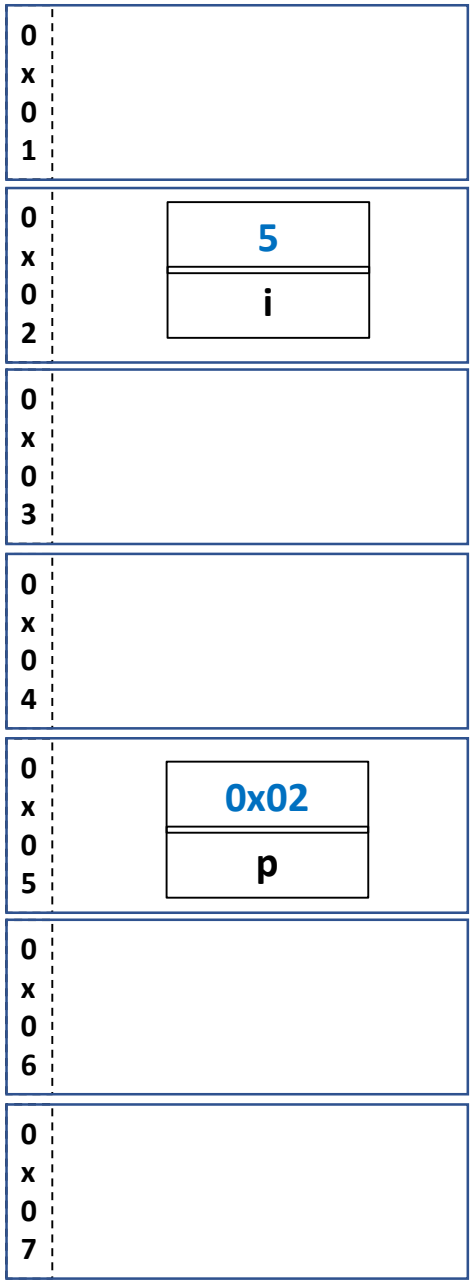


Observação importante:

```
#include <stdio.h>

int main(void){
        int i = 7;
        int *p = NULL;

        p = &i;
        *p = 5;
```

Observe essa versão mais completa e correta do programa. Apesar de ter sido iniciado com ***NULL***, foi atribuído o endereço de ***i*** para ***p*** e feito o acesso indireto para alterar o valor dessa variável. Tudo normal.

```
        return 0;
}
```



Observação importante:

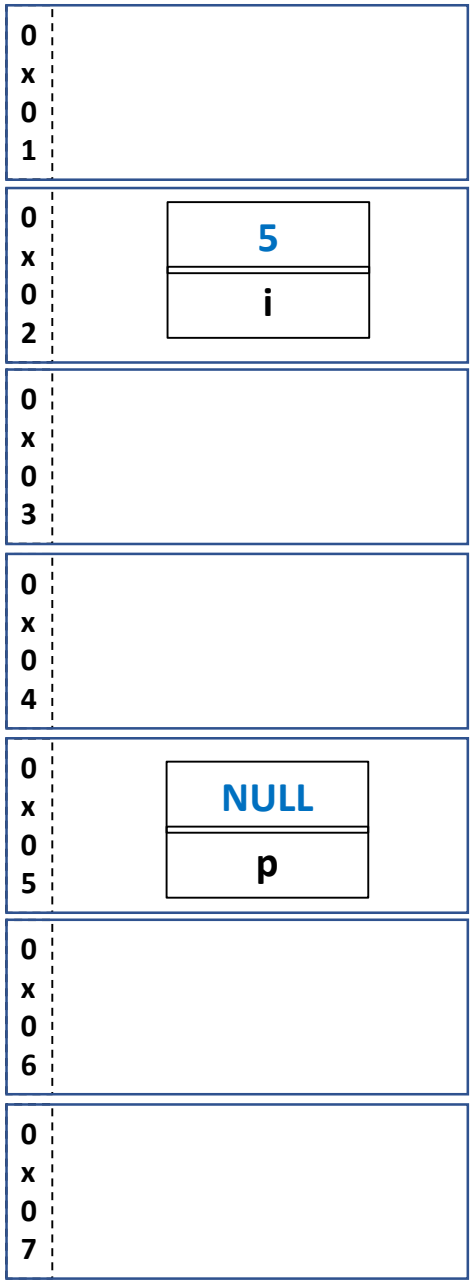
```
#include <stdio.h>

int main(void){
        int i = 7;
        int *p = NULL;

        p = &i;
        *p = 5;
        p = NULL;
```

É também uma boa prática deixar o ponteiro nulo novamente quando não mais precisamos fazer acesso indireto em uma variável.

```
        return 0;
}
```

Essa prática pode evitar erros no programa, caso seja tentado novo acesso indireto indevido no ponteiro ***p***.

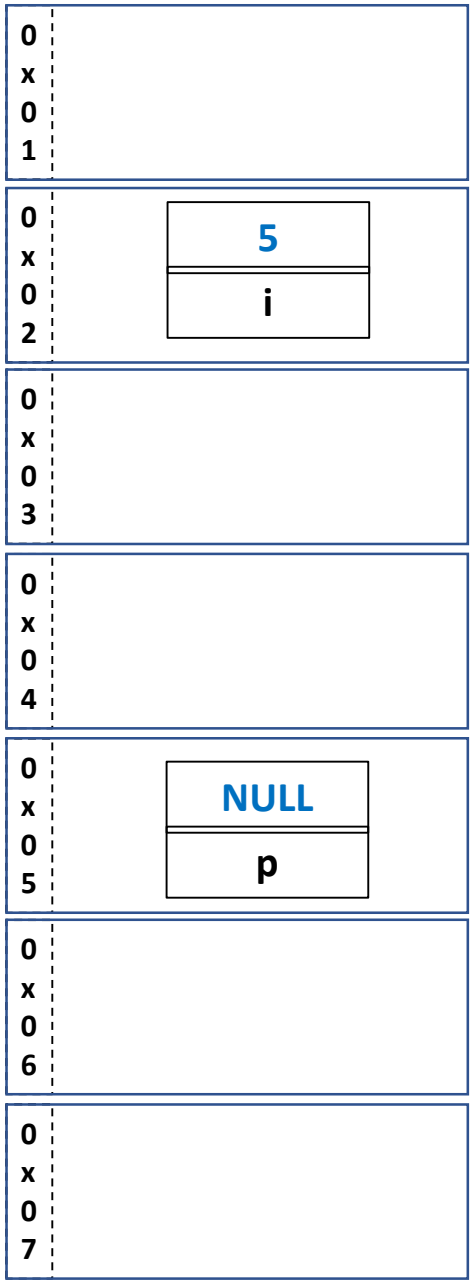
Memória



Observação importante:

Observe a versão completa do programa e verifique se entende o porquê do ***printf*** imprimir ***5*** como valor de ***i***.

```
#include <stdio.h>

int main(void){
        int i = 7;
        int *p = NULL;

        p = &i;
        *p = 5;
        p = NULL;

        printf("valor de i: %d\n", i);



        return 0;
}
```



Modos de Parâmetros

- **Parâmetros de entrada:**
  - Os valores são apenas lidos/consultados;
  - Se for ponteiro, o valor da variável apontada *não deve* ser modificado.
- **Parâmetros de saída:**
  - Deve ser ponteiro;
  - O valor da variável apontada *deve* ser modificado pela função.
- **Parâmetros entrada e saída:**
  - Deve ser ponteiro;
  - O valor da variável apontada *deve ser* consultado e também modificado.

Exercícios: Passagem de parâmetros

1. Escreva uma função chamada MM que recebe dois parâmetros, A e B, e devolve o menor dos dois em A e o maior dos dois em B. Caso sejam passados valores iguais, a ordem da resposta entre eles não importa.
2. Escreva uma função chamada medidasDoRetangulo que recebe os parâmetros b e h (as medidas em centímetros dos dois lados de um retângulo) e devolve, em parâmetros, A e P respectivamente, a área (b*h) e o perímetro (2*(b+h)) deste retângulo. Mostre os resultados na função main().
3. Altere a função anterior para que além de devolver os parâmetros A e P, respectivamente a área e o perímetro deste retângulo, retorne 1 caso o retângulo seja um quadrado, ou 0 caso contrário.

Exercícios: Passagem de parâmetros

4. Escreva uma função que receba um numero inteiro positivo representando os segundos e converta-o para horas, minutos e segundos.
5. Escreva uma função que recebe por parâmetro um valor inteiro e positivo N e retorna o valor de S. Escreva também uma função para mostrar a sequência no formato apresentado abaixo.

$$S = 2/4 + 5/5 + 10/6 + 17/7 + 26/8 + ... +(N^2+1)/(N+3)$$