

Uso de Ponteiros em C

Agostinho Brito

2022

Agenda

- 1 Para que existem os ponteiros
- 2 O que são ponteiros
- 3 Ponteiros e funções (parte I)
- 4 Navegando na memória
- 5 Alocação dinâmica de memória - vetores
- 6 Alocação dinâmica de memória - matrizes
- 7 Ponteiros e funções (parte II)
- 8 Ponteiros e *structs*
- 9 Ponteiros para funções

Para que existem os ponteiros

Para que existem os ponteiros?

Considere uma função simples destinada a trocar dois valores de lugar

```
1 #include <stdio.h>
2
3 void troca(int a, int b) {
4     int tmp;
5     printf("a = %d; b = %d\n", a, b);
6     tmp=a; a=b; b=tmp;
7     printf("a = %d; b = %d\n", a, b);
8 }
9
10 int main(void) {
11     int x=3, y=4;
12     printf("x = %d; y = %d\n", x, y);
13     troca(x, y);
14     printf("x = %d; y = %d\n", x, y);
15     return 0;
16 }
```

Para que existem os ponteiros?

Considere uma função simples destinada a trocar dois valores de lugar

```
1 #include <stdio.h>
2
3 void troca(int a, int b) {
4     int tmp;
5     printf("a = %d; b = %d\n", a, b);
6     tmp=a; a=b; b=tmp;
7     printf("a = %d; b = %d\n", a, b);
8 }
9
10 int main(void) {
11     int x=3, y=4;
12     printf("x = %d; y = %d\n", x, y);
13     troca(x, y);
14     printf("x = %d; y = %d\n", x, y);
15     return 0;
16 }
```

Saída

x = 3; y = 4
a = 3; b = 4
a = 4; b = 3
x = 3; y = 4

Para que existem os ponteiros?

Considere uma função simples destinada a trocar dois valores de lugar

```
1 #include <stdio.h>
2
3 void troca(int a, int b) {
4     int tmp;
5     printf("a = %d; b = %d\n", a, b);
6     tmp=a; a=b; b=tmp;
7     printf("a = %d; b = %d\n", a, b);
8 }
9
10 int main(void) {
11     int x=3, y=4;
12     printf("x = %d; y = %d\n", x, y);
13     troca(x, y);
14     printf("x = %d; y = %d\n", x, y);
15     return 0;
16 }
```

Saída

x = 3; y = 4
a = 3; b = 4
a = 4; b = 3
x = 3; y = 4

Porquê?

Passagem de parâmetros por valor.

Para que existem os ponteiros?

Considere uma função simples destinada a trocar dois valores de lugar

```
1 #include <stdio.h>
2
3 void troca(int a, int b) {
4     int tmp;
5     printf("a = %d; b = %d\n", a, b);
6     tmp=a; a=b; b=tmp;
7     printf("a = %d; b = %d\n", a, b);
8 }
9
10 int main(void) {
11     int x=3, y=4;
12     printf("x = %d; y = %d\n", x, y);
13     troca(x, y);
14     printf("x = %d; y = %d\n", x, y);
15     return 0;
16 }
```

Saída

x = 3; y = 4
a = 3; b = 4
a = 4; b = 3
x = 3; y = 4

Porquê?

Passagem de parâmetros por valor.

Solução?

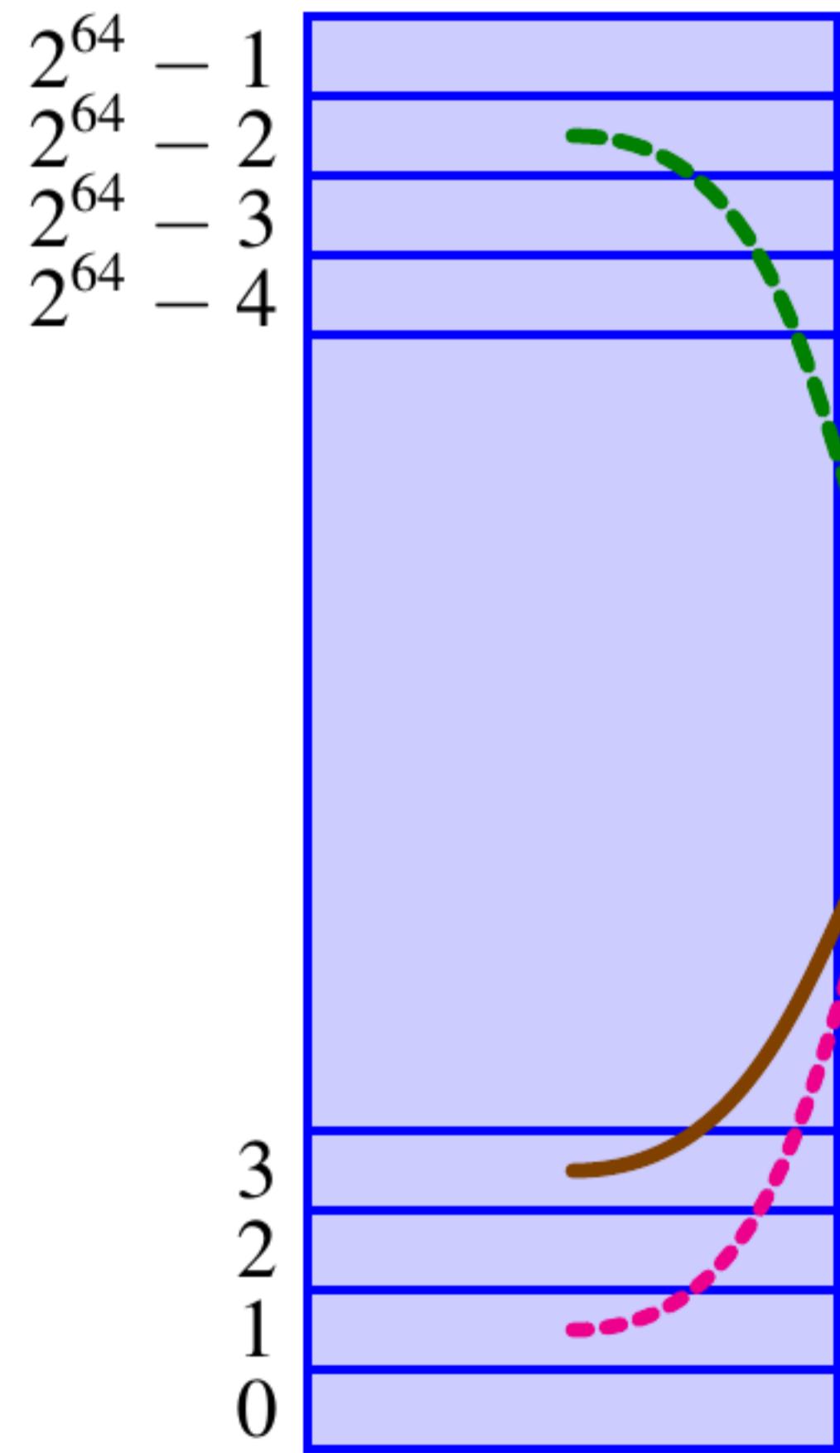
Uso de ponteiros.

Organização da memória na máquina

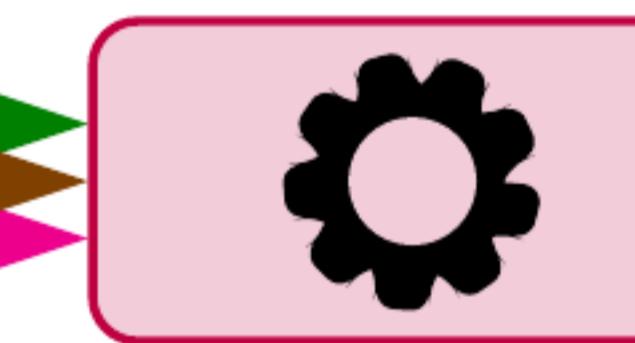
- Na máquina, a organização da memória exerce um papel fundamental na forma como os programas executam.
- Nos programas de computador, a estrutura de memória é provida pelo sistema operacional, sendo organizada na chamada **memória virtual**.
- A memória virtual é um recurso usado para unificar a forma como os programas acessam os tipos de memória que a máquina possui: principal(RAM) e secundária(discos).

A memória virtual

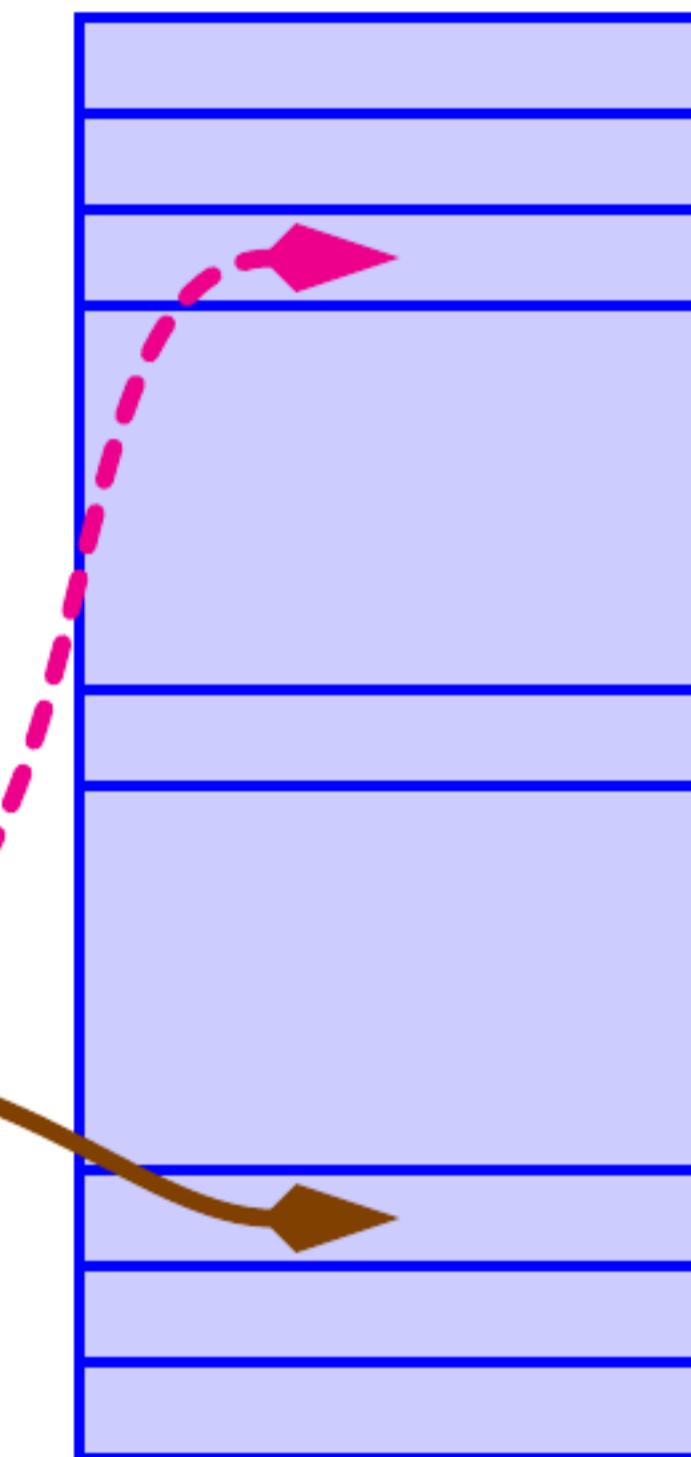
Memória Virtual 64 bits
(2^{64} bytes possíveis)



Mapeador de
Memória



Memória Principal
Ex: 8GB RAM

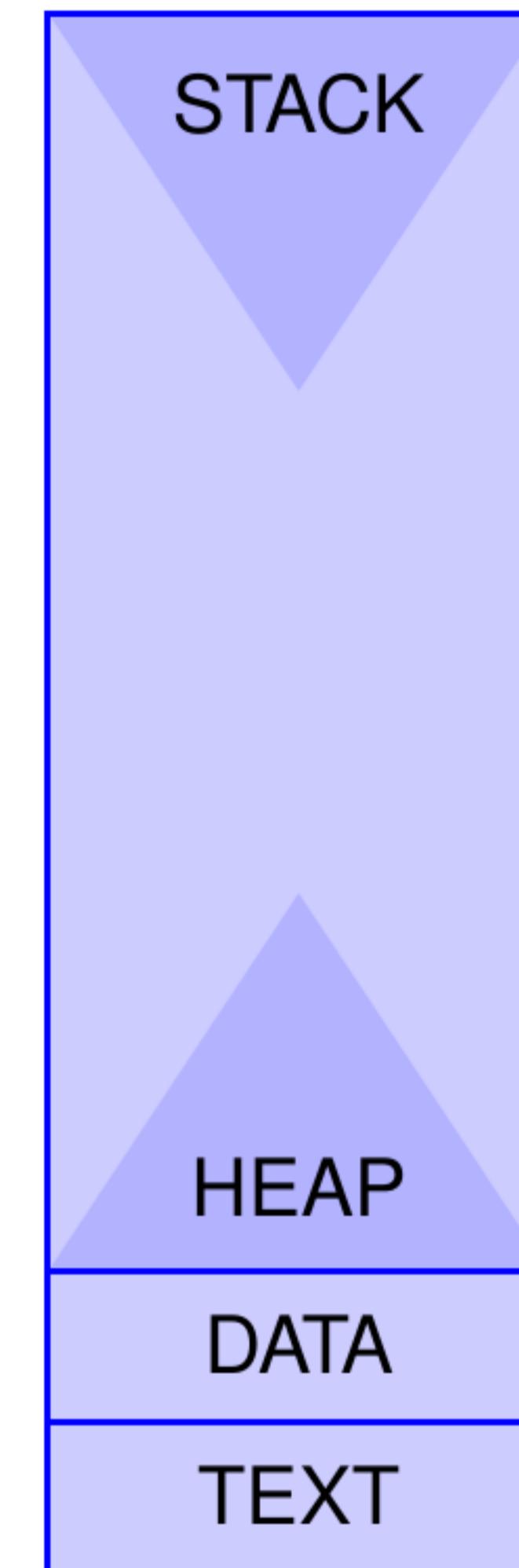


Disco Rígido

A memória virtual

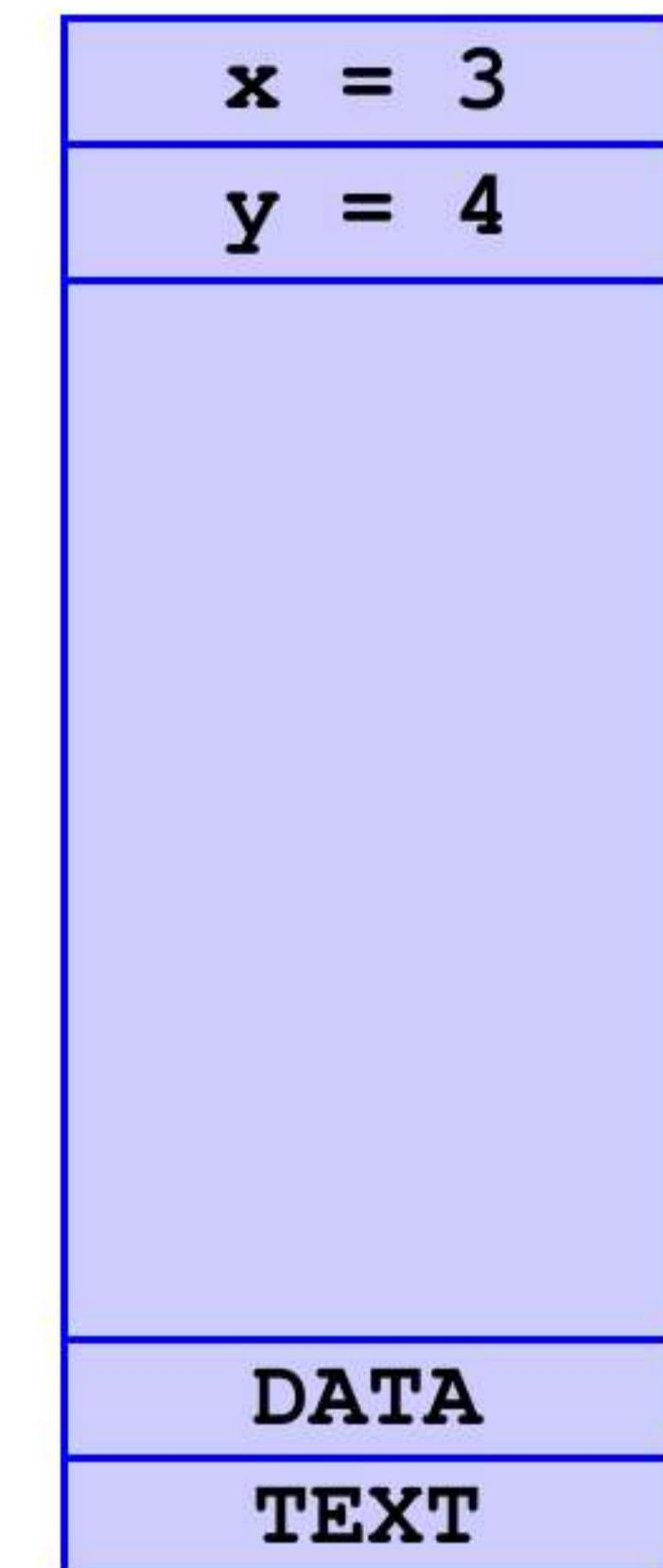
- Os compiladores organizam o código para usar a memória virtual como uma forma de unificar para as diversas arquiteturas à estrutura de memória que o programa necessita para operar.
- A memória virtual é normalmente organizada em segmentos, que agregam as informações do programa conforme sua utilidade.
- Arquiteturas tradicionais utilizam quatro segmentos: texto (ou código), dados, *heap* e *stack*.

Memória Virtual



Usando a memória virtual

```
1 void troca(int a, int b) {  
2     int tmp=0;  
3     printf("a = %d; b = %d\n", a, b);  
4     tmp=a; a=b; b=tmp;  
5     printf("a = %d; b = %d\n", a, b);  
6 }  
7 int main(void) {  
8     int x=3, y=4;  
9     printf("x = %d; y = %d\n", x, y);  
10    troca(x, y);  
11    printf("x = %d; y = %d\n", x, y);  
12    return 0;  
13 }
```



Usando a memória virtual

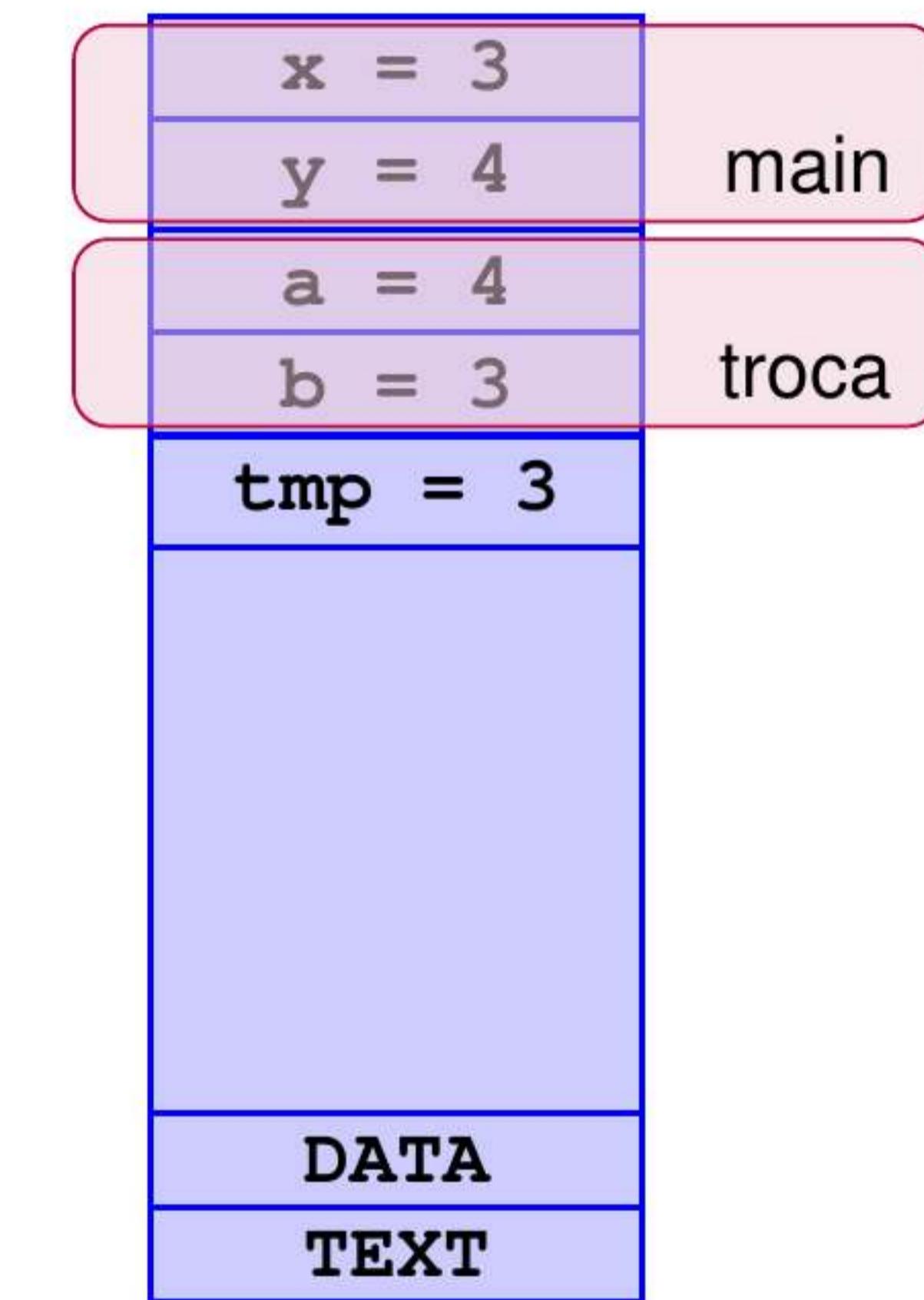
```
1 void troca(int a, int b) {  
2     int tmp=0;  
3     printf("a = %d; b = %d\n", a, b);  
4     tmp=a; a=b; b=tmp;  
5     printf("a = %d; b = %d\n", a, b);  
6 }  
7 int main(void) {  
8     int x=3, y=4;  
9     printf("x = %d; y = %d\n", x, y);  
10    troca(x, y);  
11    printf("x = %d; y = %d\n", x, y);  
12    return 0;  
13 }
```

x = 3
y = 4
a = 3
b = 4
tmp = 0



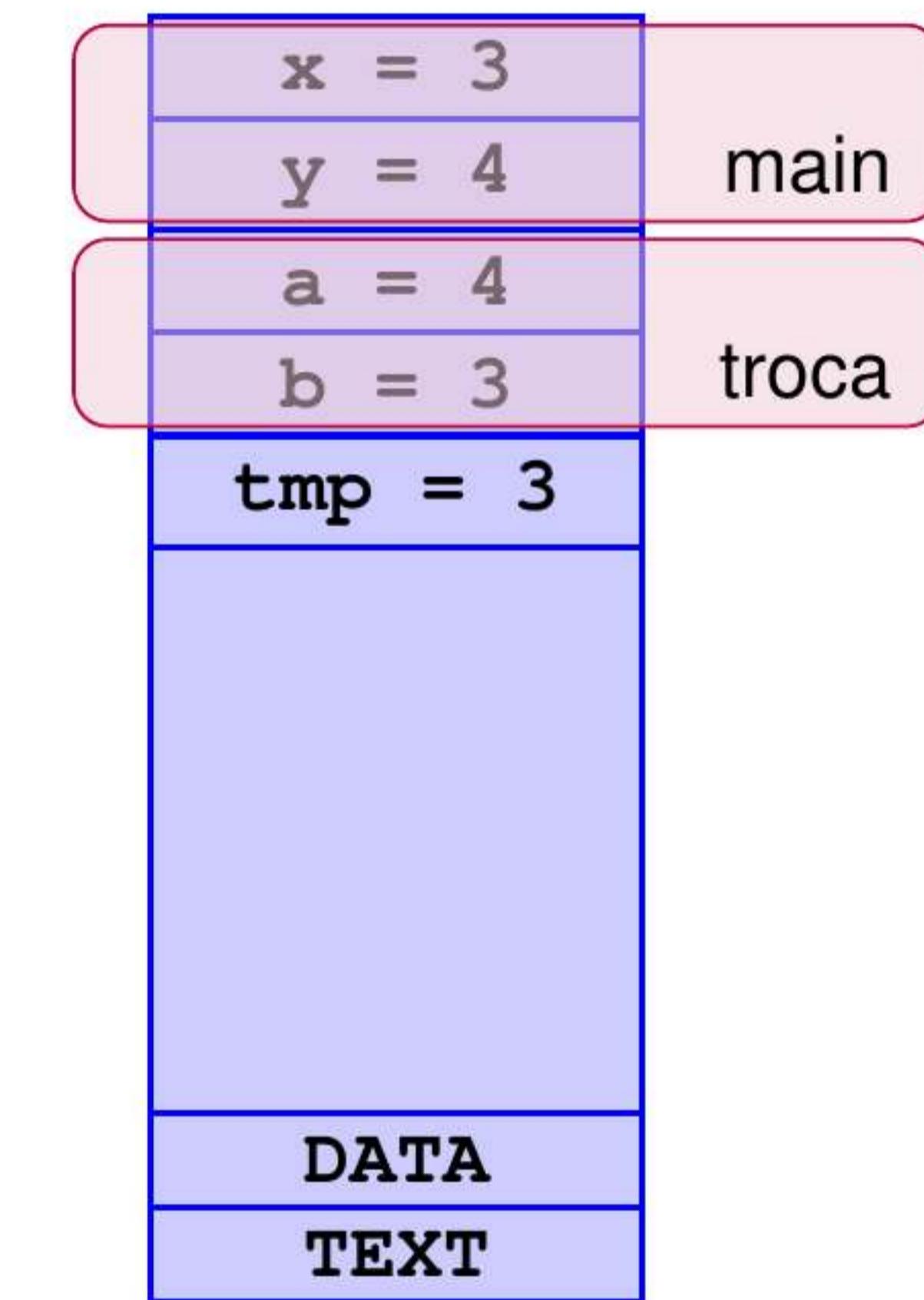
Usando a memória virtual

```
1 void troca(int a, int b) {  
2     int tmp=0;  
3     printf("a = %d; b = %d\n", a, b);  
4     tmp=a; a=b; b=tmp;  
5     printf("a = %d; b = %d\n", a, b);  
6 }  
7 int main(void) {  
8     int x=3, y=4;  
9     printf("x = %d; y = %d\n", x, y);  
10    troca(x, y);  
11    printf("x = %d; y = %d\n", x, y);  
12    return 0;  
13 }
```



Usando a memória virtual

```
1 void troca(int a, int b) {  
2     int tmp=0;  
3     printf("a = %d; b = %d\n", a, b);  
4     tmp=a; a=b; b=tmp;  
5     printf("a = %d; b = %d\n", a, b);  
6 }  
7 int main(void) {  
8     int x=3, y=4;  
9     printf("x = %d; y = %d\n", x, y);  
10    troca(x, y);  
11    printf("x = %d; y = %d\n", x, y);  
12    return 0;  
13 }
```



Conclusão

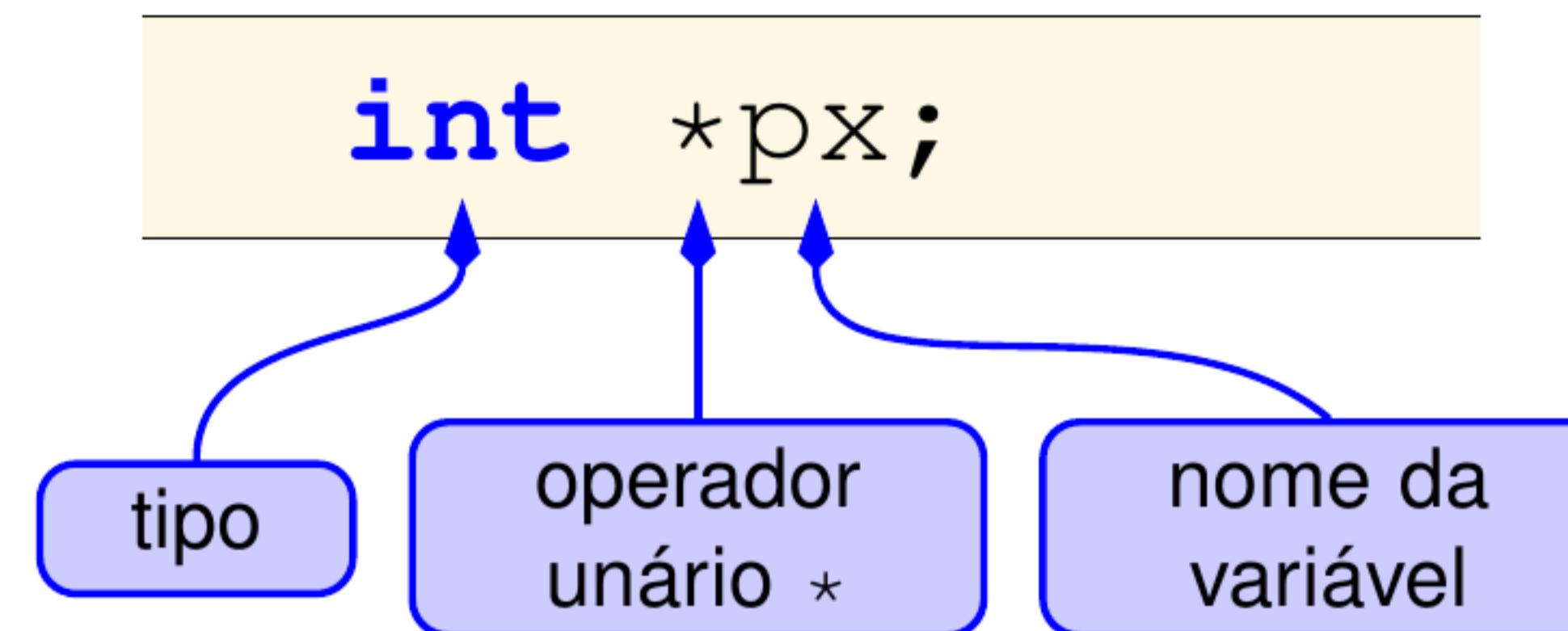
A troca não ocorre nas variáveis x e y, pois a passagem de argumentos é feita **POR VALOR!**

O que são ponteiros

O que são ponteiros?

- Ponteiros são tipos especiais de variáveis usadas para armazenar **endereços de memória**.
- Para cada tipo diferente de variável é necessário um tipo de ponteiro específico.
- Um ponteiro ocupa, em bytes, a quantidade em bytes suficiente para endereçar todas as possíveis posições de memória que se usa em uma arquitetura específica de sistema operacional.
- **Para um S.O. de 32 bits**, os endereços de memória tem 32 bits de tamanho, de modo que uma variável ponteiro nessa arquitetura ocupará exatamente **4 bytes**.
- Diferentemente dos tipos de dados regulares, que podem possuir tamanhos diferentes, os ponteiros possuem sempre o mesmo tamanho em uma arquitetura específica.

Declarando ponteiros



- Na linha acima, diz-se que `px` é uma variável do tipo **ponteiro para int**.
- A variável `px` é capaz de armazenar o endereço onde algum inteiro se encontra na memória.
- Para um tipo diferente de `int`, declara-se de forma semelhante:

```
TIPO *ponteiro;
```

- Quaisquer identificadores podem ser atribuídos a uma variável ponteiro, desde que sigam às mesmas regras para as variáveis regulares.

```
1 int *px, x, y;  
2 x = 2;  
3 px = &x;  
4 y = *px;
```

Descobrindo endereços

O símbolo **&**, quando colocado antes de uma variável qualquer, retorna o **ENDEREÇO** desta variável. É chamado de **operador de endereço** (*address operator*).

Descobrindo conteúdos

O símbolo *****, quando colocado antes de uma variável **PONTEIRO**, recupera o **CONTEÚDO DO ENDEREÇO** armazenado no ponteiro. É chamado de **operador de dereferência** (*dereference operator*).

Ponteiros apontam para variáveis

```
int x, *px = &x;
```



Diz-se que **px** aponta para **x**

- Endereços em C também são impressos usando a função `printf()`.
- O string de formato normalmente usado para esse fim é `%p` (abbreviatura de *pointer*).
- O endereço é mostrado em base **hexadecimal**.

```
1 int x, *px;  
2 px = &x;  
3 printf("Valor de px = %p", px);
```

- Os endereços também podem ser impressos como tipos inteiros. Entretanto, deve-se prestar atenção ao tipo inteiro usado, que pode ser de 32 bits ou 64 bits, dependendo da arquitetura onde o programa executa.

Operando com ponteiros

- Para modificar o valor armazenado em ponteiro, precisa-se usar o operador de dereferenciação.

```
1 int x, *px = &x;
2 *px = 3;    // x recebe o valor 3
3 *px = *px + 1; // px recebe o valor 4
4 (*px)--; // x retorna ao valor 3
```



Obrigado

Ponteiros e funções (parte I)



Praticando a troca de variáveis...

```
1 #include <stdio.h>
2
3 void troca(int *a, int *b) {
4     int *tmp;
5     printf("a = %d; b = %d\n", *a, *b);
6     tmp=*a; *a=*b; *b=tmp;
7     printf("a = %d; b = %d\n", *a, *b);
8 }
9
10 int main(void) {
11     int x=3, y=4;
12     printf("x = %d; y = %d\n", x, y);
13     troca(&x, &y);
14     printf("x = %d; y = %d\n", x, y);
15     return 0;
16 }
```



Praticando a troca de variáveis...

```
1 #include <stdio.h>
2
3 void troca(int *a, int *b) {
4     int *tmp;
5     printf("a = %d; b = %d\n", *a, *b);
6     tmp=*a; *a=*b; *b=tmp;
7     printf("a = %d; b = %d\n", *a, *b);
8 }
9
10 int main(void) {
11     int x=3, y=4;
12     printf("x = %d; y = %d\n", x, y);
13     troca(&x, &y);
14     printf("x = %d; y = %d\n", x, y);
15     return 0;
16 }
```



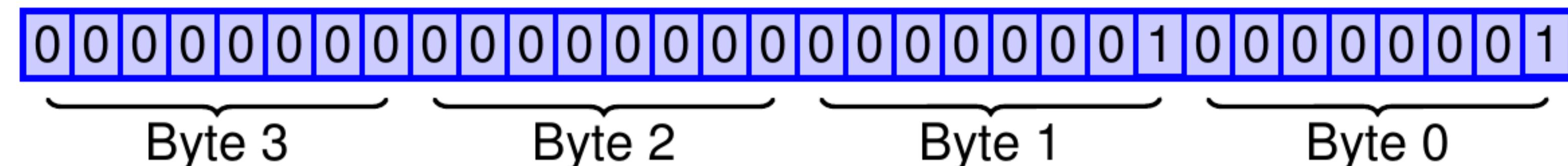
Obrigado

Navegando na memória

Navegando com ponteiros na memória virtual

- Um dos principais usos dos ponteiros é percorrer a memória da máquina a partir de um endereço de base.
- Isso é feito incrementando ou decrementando os ponteiros.
- As operações permitidas com ponteiros são as de incremento e decrecimento com valores **INTEIROS**.
- Quatro operadores podem ser usados com ponteiros: `++`, `--`, `+` e `-`.
- A cada incremento (ou decrecimento), o ponteiro caminha **N bytes**, conforme seja o tamanho do tipo de dado para o qual ele aponta.
- Por exemplo, para um tipo de dado `int` que ocupe 4 bytes, um ponteiro para `int` (`px`), que guardasse o endereço `A` apontaria para o endereço `A+4` caso sofresse a operação `px++`;

Explorando uma variável `unsigned int` de 32 bits



```
1 unsigned int x;  
2 unsigned char *px;  
3 px = x;  
4 printf("%d\n", *px);
```

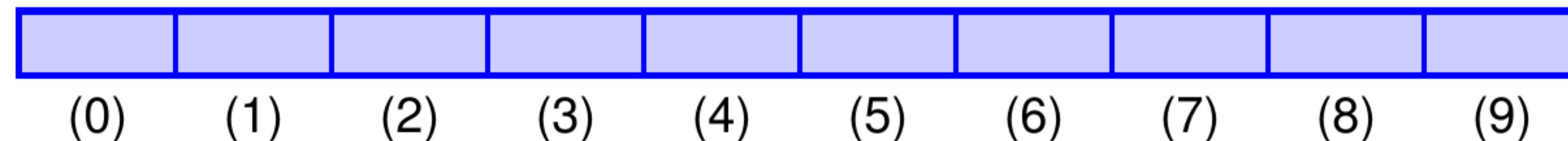
- ➊ Declarar a variável inteira de 32 bits, `x`.
- ➋ Declarar um ponteiro do tipo *unsigned char*, `px`. Ponteiros dessa natureza caminham byte a byte na memória quando são incrementados ou decrementados.
- ➌ Fazer `px` apontar para `x`.



Praticando a caminhada na memória...

Arrays e ponteiros

- Arrays, arranjos ou vetores são um tipo de estrutura indexada capaz de guardar um conjunto determinado de elementos de um mesmo tipo.
- Sua representação é na forma de uma sequência, onde os elementos ficam consecutivos uns aos outros na memória.



- Ao utilizar ponteiros, os incrementos (ou decrementos) realizados nestes permitem dar **saltos** na memória de tamanho igual à quantidade de bytes ocupada pelo tipo apontado pelo ponteiro.
- Pode-se usar um ponteiro para armazenar o endereço de um array e realizar operações neste array.

```
int x[10];
int *px = x;
```



Praticando Arrays e Ponteiros...



Obrigado

Alocação dinâmica de memória - vetores

Alocação dinâmica de memória

- Arrays permitem alocar uma coleção de elementos... **mas de tamanho fixo.**
- E se precisássemos de alocar um array cujo tamanho fosse definido em tempo de execução?

- A linguagem C provê funções para **ALOCAR** e **LIBERAR** memória em tempo de execução.
- Esse recurso permite que o programador gerencie sua própria memória, alocando **EXATAMENTE** a quantidade necessária de espaço que precisa para armazenar seu conjunto de dados.
- Uma vez alocado, o recurso pode ser utilizado da mesma forma que um array.
- Para usar alocação, é necessário usar ponteiros para guardar os endereços dos blocos alocados... e liberá-los quando não são mais necessários.

Alocação dinâmica de memória

- Arrays permitem alocar uma coleção de elementos... **mas de tamanho fixo.**
- E se precisássemos de alocar um array cujo tamanho fosse definido em tempo de execução?

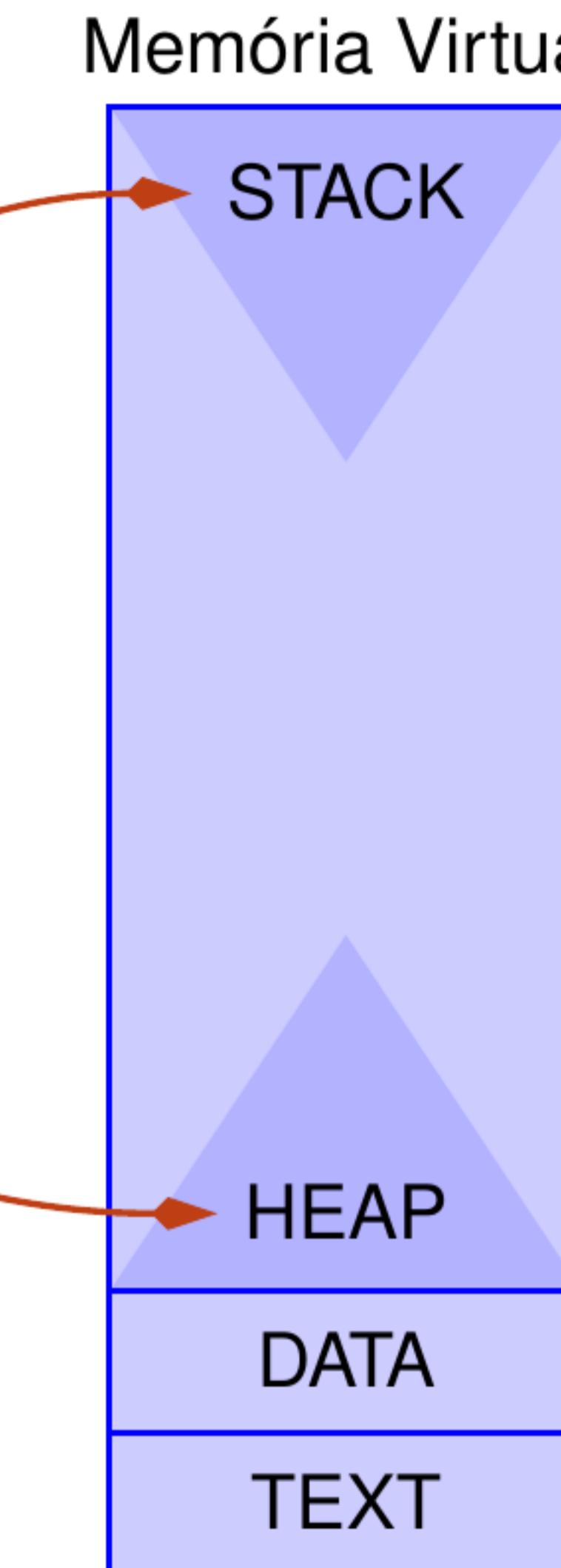
Solução

Alocação dinâmica de memória.

- A linguagem C provê funções para **ALOCAR** e **LIBERAR** memória em tempo de execução.
- Esse recurso permite que o programador gerencie sua própria memória, alocando **EXATAMENTE** a quantidade necessária de espaço que precisa para armazenar seu conjunto de dados.
- Uma vez alocado, o recurso pode ser utilizado da mesma forma que um array.
- Para usar alocação, é necessário usar ponteiros para guardar os endereços dos blocos alocados... e liberá-los quando não são mais necessários.

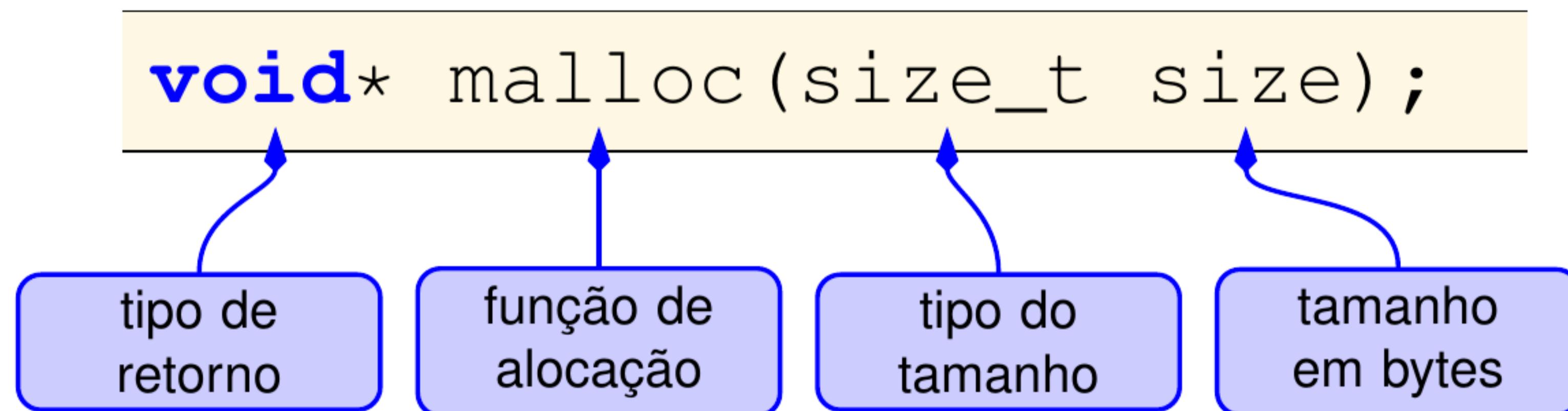
Alocação dinâmica de memória

- A alocação de memória dinâmica ocorre de forma diferente daquela usada para alocar arrays.
- Arrays são alocados no **stack**.
- Alocação dinâmica de memória se dá no **heap**.



Alocando memória com `malloc()`

- A biblioteca padrão do C possui algumas funções para alocação dinâmica de memória definidas no *header* `stdlib.h`.
- A função `malloc()` é a mais popular.



- `size_t` é um tipo de dado previsto na linguagem C para denotar tamanhos de variáveis, ocupando pelo menos 16 bits de memória.
- O tipo de retorno da função `malloc()` é sempre `void*` (ponteiro para `void`).

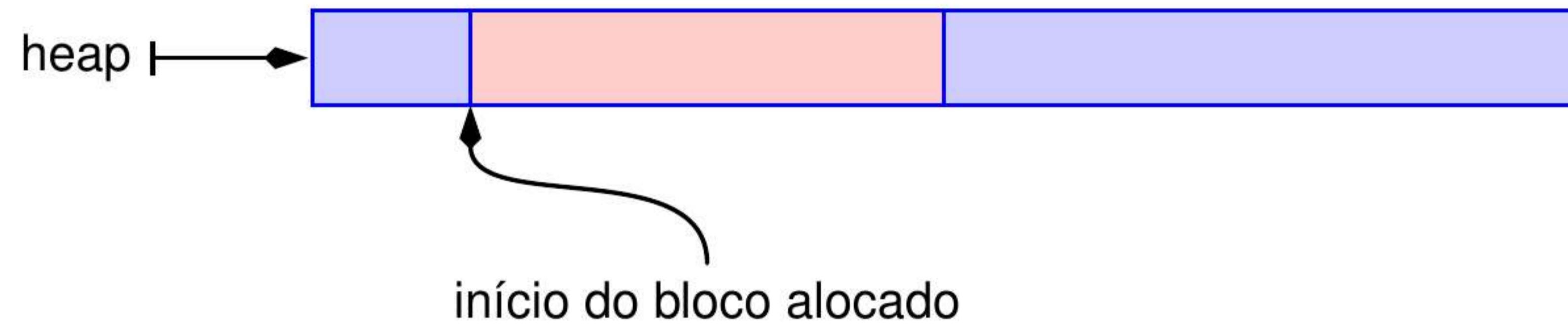
- Alocando um bloco de 10 inteiros:

```
int *x;  
x = malloc(10 * sizeof(int));
```

- A alocação normalmente é feita calculando a quantidade necessária (em bytes) para armazenar o bloco de dados.
- O uso do **operador** `sizeof()` permite descobrir quantos bytes um determinado tipo ocupa na memória.
- No exemplo, invoca-se `malloc()` passando o produto da quantidade de elementos pelo tamanho em bytes de cada um.

Como saber se a alocação funcionou?

- A função `malloc()` retorna o **ENDEREÇO** do primeiro byte do bloco alocado. Caso não seja alocado, o valor “0” (zero) é retornado.
- O valor “0” também é conhecido como ponteiro **NULL**.
- Caso a alocação seja bem sucedida, o endereço é armazenado na variável `x`.



Sempre atribua valor NULL a ponteiros não utilizados.

Usando a memória alocada

- Uma vez alocado um bloco de memória, o mesmo torna-se acessível através da variável que aponta para este.

```
* (x+1) = 3;
```

... ou melhor ainda...

```
x [1] = 3;
```



Praticando alocação dinâmica...

- Os blocos de memória alocados por `malloc()` permanecem reservados até que sejam liberados com a função `free()` ou quando o programa finaliza.
- Blocos de memórias não utilizados devem ser liberados para evitar **vazamentos de memória**.

REGRA DE OURO

Para cada `malloc()` deve haver um `free()`

Liberando memória com `free()`

- Os blocos de memória alocados por `malloc()` permanecem reservados até que sejam liberados com a função `free()` ou quando o programa finaliza.
- Blocos de memórias não utilizados devem ser liberados para evitar **vazamentos de memória**.

REGRA DE OURO

Para cada `malloc()` deve haver um `free()`

3 Regras simples para liberar espaços de memória

- ① Não acesse espaços de memória que já foram liberados, pois os blocos de memória já não pertencem ao programa.
- ② Libere os espaços de memória na ordem inversa em que foram alocados. Dependências entre blocos de memória não geram ponteiros perdidos.
- ③ Sempre que sair de uma subrotina, libere os blocos de memória temporários.

- Os blocos de memória alocados por `malloc()` permanecem reservados até que sejam liberados com a função `free()` ou quando o programa finaliza.
- Blocos de memórias não utilizados devem ser liberados para evitar **vazamentos de memória**.

REGRA DE OURO

Para cada `malloc()` deve haver um `free()`

3 Regras simples para liberar espaços de memória

- ① Não acesse espaços de memória que já foram liberados, pois os blocos de memória já não pertencem ao programa.
- ② Libere os espaços de memória na ordem inversa em que foram alocados. Dependências entre blocos de memória não geram ponteiros perdidos.
- ③ Sempre que sair de uma subrotina, libere os blocos de memória temporários.

- Os blocos de memória alocados por `malloc()` permanecem reservados até que sejam liberados com a função `free()` ou quando o programa finaliza.
- Blocos de memórias não utilizados devem ser liberados para evitar **vazamentos de memória**.

REGRA DE OURO

Para cada `malloc()` deve haver um `free()`

3 Regras simples para liberar espaços de memória

- ① Não acesse espaços de memória que já foram liberados, pois os blocos de memória já não pertencem ao programa.
- ② Libere os espaços de memória na ordem inversa em que foram alocados. Dependências entre blocos de memória não geram ponteiros perdidos.
- ③ Sempre que sair de uma subrotina, libere os blocos de memória temporários.

- A função `free()` recebe como parâmetro o endereço do bloco a ser liberado.
- **NUNCA** forneça à função `free()` um endereço de um bloco que já fora liberado ou que não esteja associado a um bloco alocado. Nestes casos, o programa finaliza por erro.
- Exemplo de bloco completo com alocação/uso/liberação de memória.

```
1 int *x;  
2 x = malloc(10 * sizeof(int));  
3 x[1] = 3;  
4 free(x);
```



Praticando `malloc()`/`free()` ...



Obrigado

Alocação dinâmica de memória - matrizes

- Matrizes declaradas da forma “`int x[4][5]`” são alocadas no *stack* junto com as outras variáveis locais. A variável `x` que representa a matriz guarda o endereço do primeiro elemento da primeira linha da matriz.
- Matrizes alocadas no *stack* são comumente chamadas de *Arrays*.
- As linhas da matriz são armazenadas uma após a outra, em sequência, na memória virtual. (outras linguagens de programação, como FORTRAN, armazenam coluna após coluna)
- Em C, matrizes alocadas dinamicamente são operadas usando **ponteiros multidimensionais**.

```
int **matriz;
```

- Diz-se que a variável `matriz` é um **ponteiro para ponteiro** para `int`, ou um ponteiro de segundo nível.

- É possível usar ponteiros multidimensionais para acessar elementos de uma matriz dinâmica da mesma forma que uma matriz alocada no *stack*.
 - Mas... quantos níveis de ponteiros podemos ter?
-
- Matrizes podem ser criadas usando hierarquias de ponteiros, mas geralmente é importante usar um esquema que permita usar as matrizes alocadas no *heap* da mesma forma que as alocadas no *stack*.

- É possível usar ponteiros multidimensionais para acessar elementos de uma matriz dinâmica da mesma forma que uma matriz alocada no *stack*.
- Mas... quantos níveis de ponteiros podemos ter?

Número mínimo de níveis

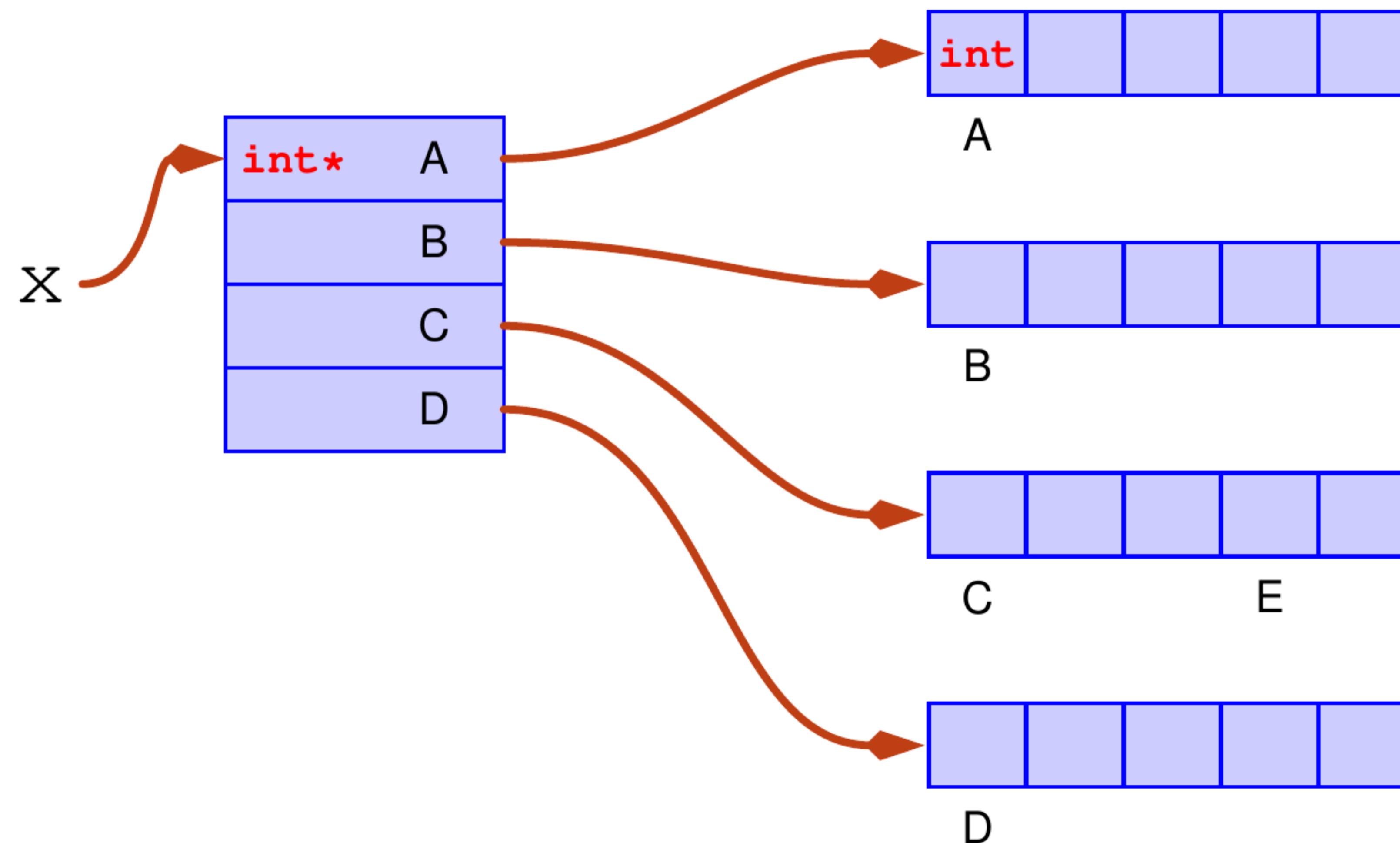
De acordo com padrão ANSI C, o compilador deve prever pelo menos 12 níveis de ponteiros.

- Matrizes podem ser criadas usando hierarquias de ponteiros, mas geralmente é importante usar um esquema que permita usar as matrizes alocadas no *heap* da mesma forma que as alocadas no *stack*.

Criando uma matriz bidimensional (parte I)

Preparando uma matriz com 4 linhas e 5 colunas

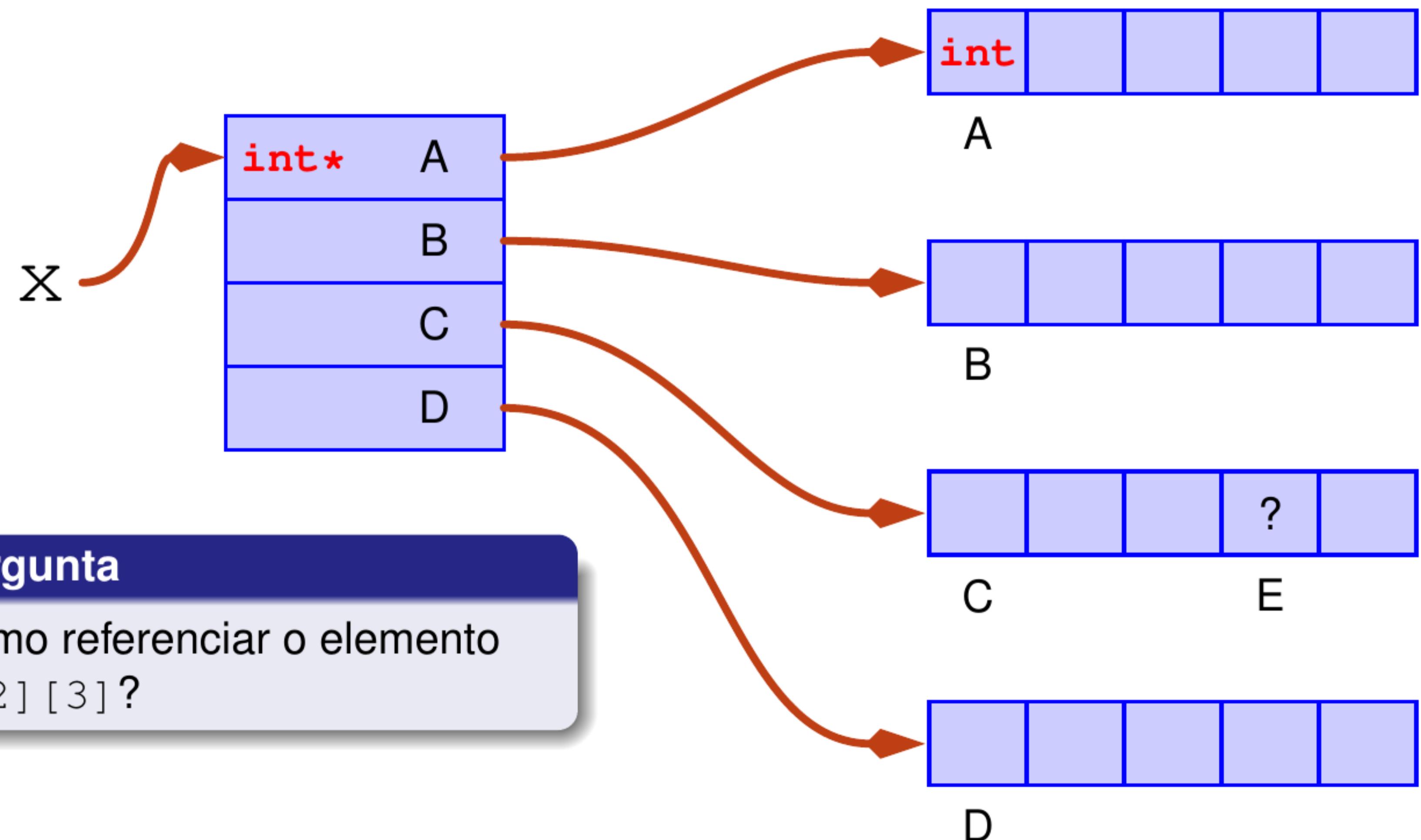
```
int **x;
```



Criando uma matriz bidimensional (parte I)

Preparando uma matriz com 4 linhas e 5 colunas

```
int **x;
```



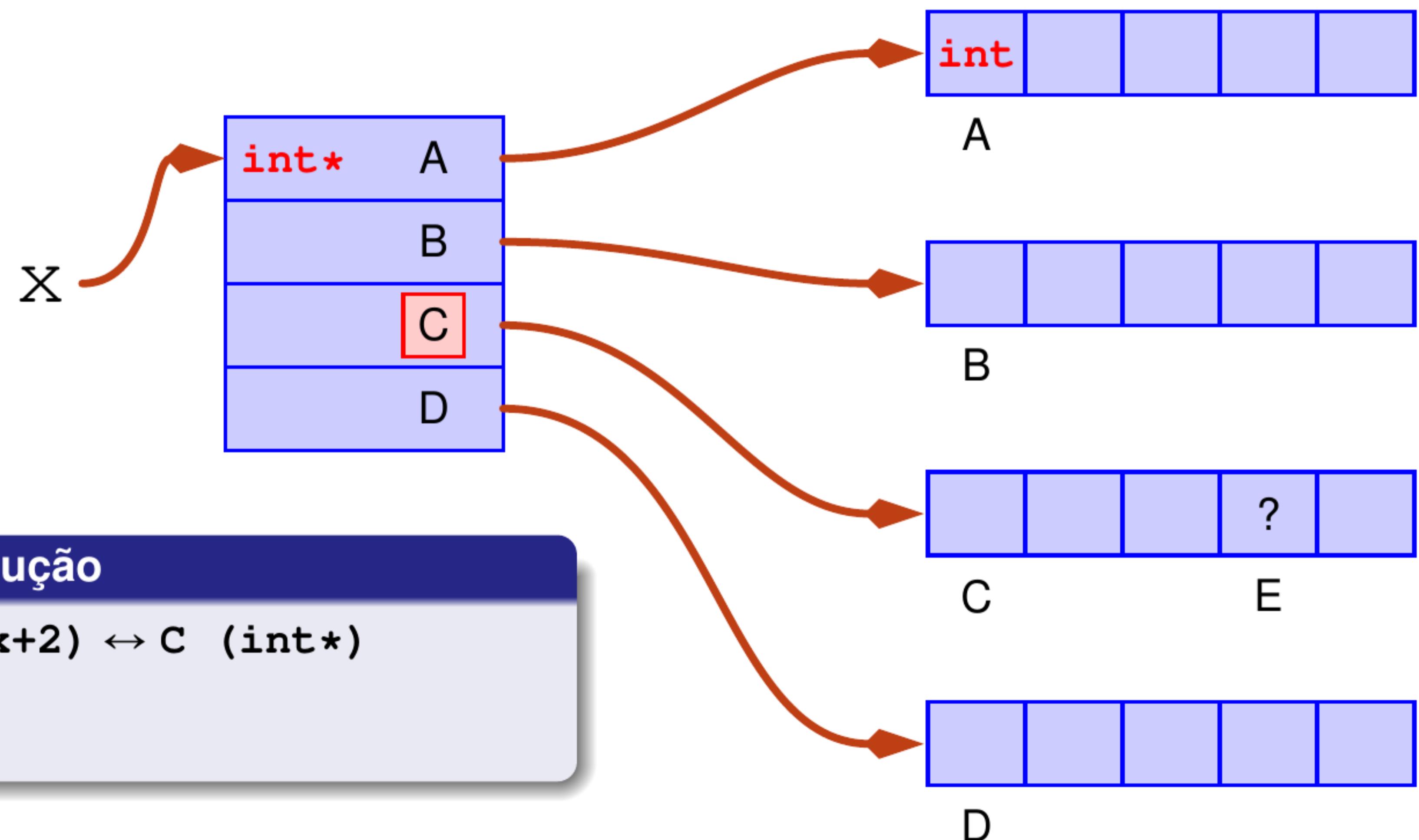
Pergunta

Como referenciar o elemento
`x[2][3]`?

Criando uma matriz bidimensional (parte I)

Preparando uma matriz com 4 linhas e 5 colunas

```
int **x;
```



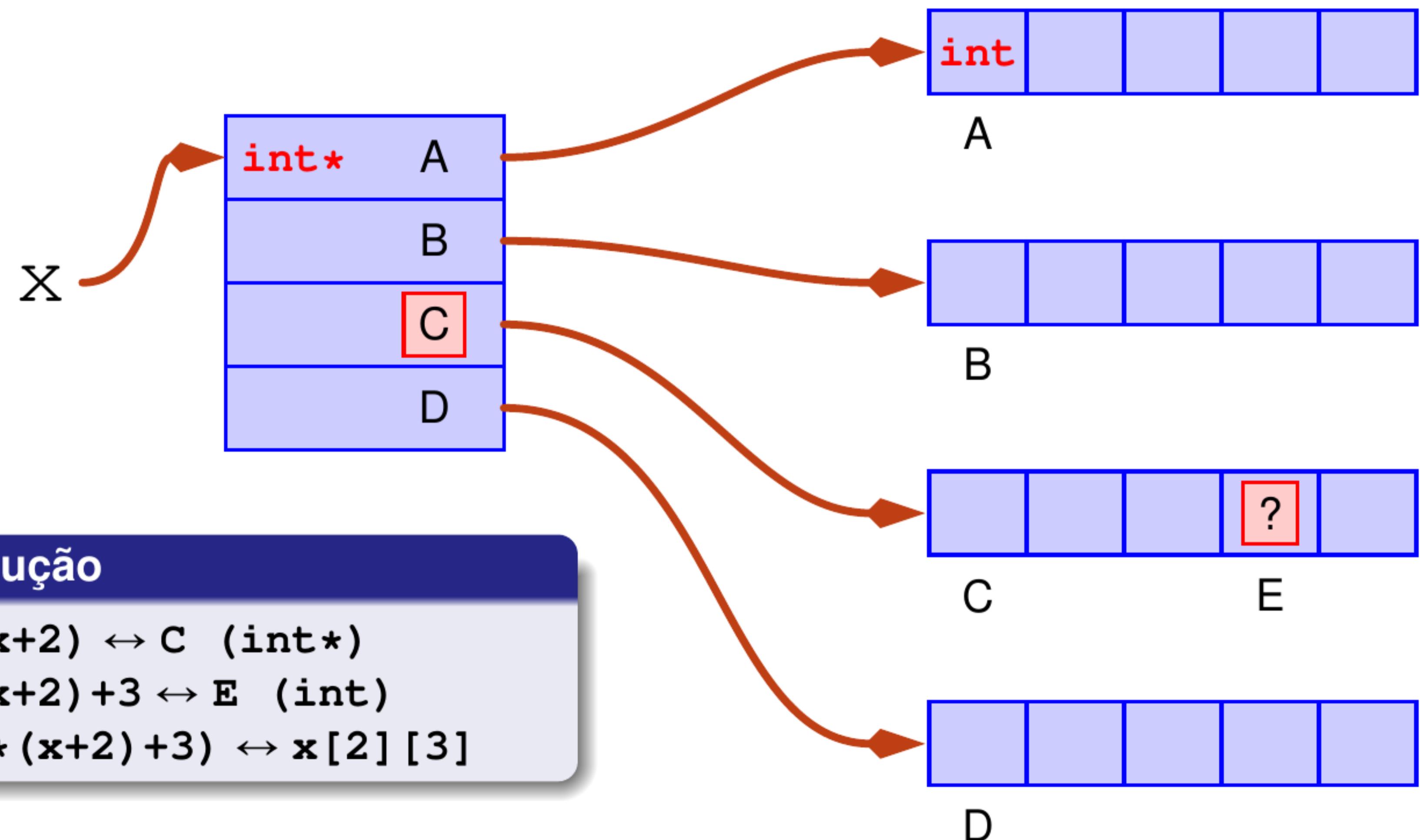
Solução

```
* (x+2) <- C (int*)
```

Criando uma matriz bidimensional (parte I)

Preparando uma matriz com 4 linhas e 5 colunas

```
int **x;
```



Solução

```
* (x+2) <- C (int*)
* (x+2)+3 <- E (int)
* (* (x+2)+3) <- x[2][3]
```

Criando uma matriz bidimensional (parte I)

```
1 #include <stdlib.h>
2
3 int main(void) {
4     int nl=4, nc=5, i;
5     int **x;
6     x = malloc(nl * sizeof(int*));
7     for(i=0; i<nl; i++) {
8         x[i] = malloc(nc * sizeof(int));
9     }
10    x[1][2]=3;
11    for(i=0; i<nl; i++) {
12        free (x[i]);
13    }
14    free (x);
15 }
```

Criando uma matriz bidimensional (parte I)

Aloca bloco auxiliar

```
1 #include <stdlib.h>
2
3 int main(void) {
4     int nl=4, nc=5, i;
5     int **x;
6     x = malloc(nl * sizeof(int*));
7     for(i=0; i<nl; i++) {
8         x[i] = malloc(nc * sizeof(int));
9     }
10    x[1][2]=3;
11    for(i=0; i<nl; i++) {
12        free (x[i]);
13    }
14    free (x);
15 }
```

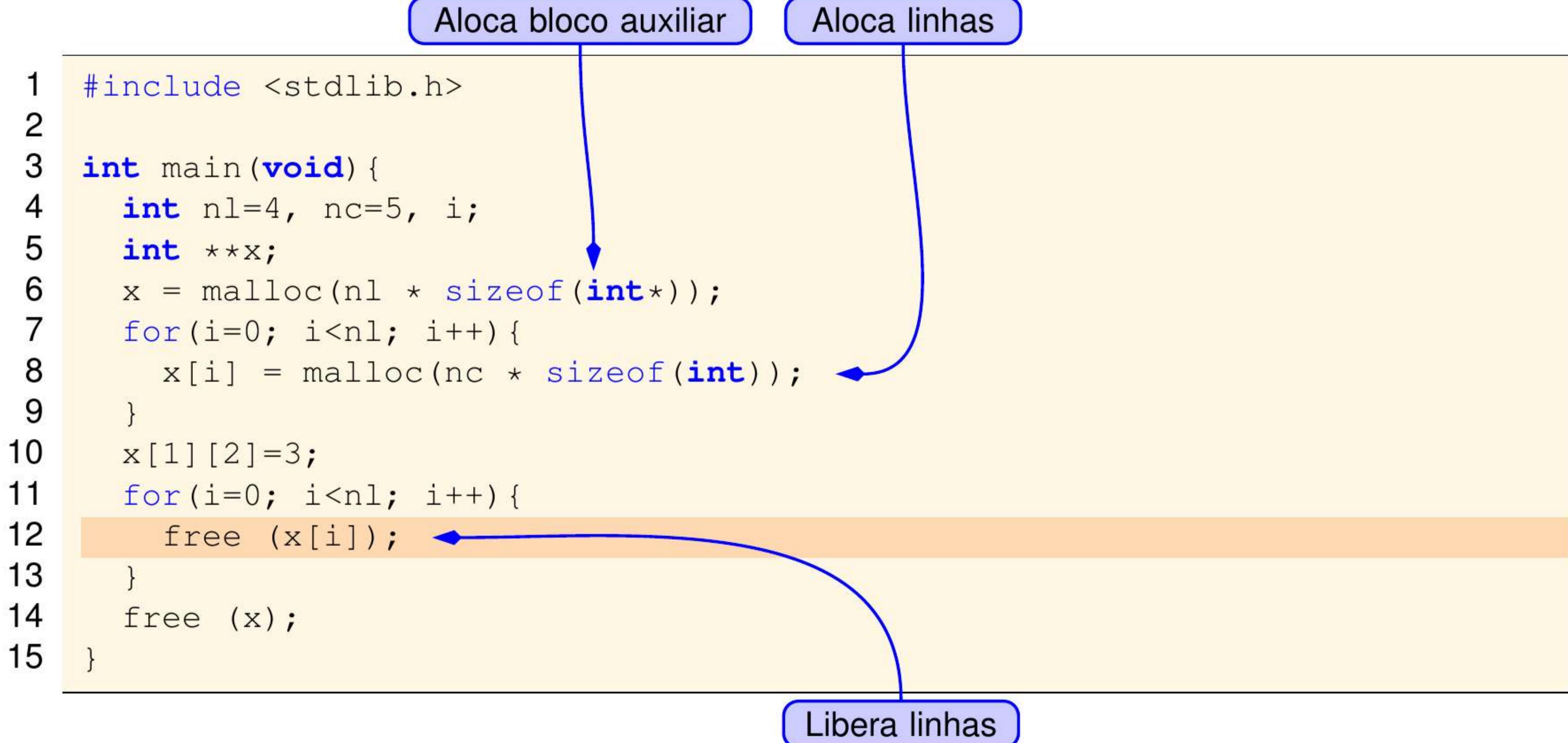
Criando uma matriz bidimensional (parte I)

Aloca bloco auxiliar

Aloca linhas

```
1 #include <stdlib.h>
2
3 int main(void) {
4     int nl=4, nc=5, i;
5     int **x;
6     x = malloc(nl * sizeof(int*));
7     for(i=0; i<nl; i++) {
8         x[i] = malloc(nc * sizeof(int));
9     }
10    x[1][2]=3;
11    for(i=0; i<nl; i++) {
12        free (x[i]);
13    }
14    free (x);
15 }
```

Criando uma matriz bidimensional (parte I)



Criando uma matriz bidimensional (parte I)

```
1 #include <stdlib.h>
2
3 int main(void) {
4     int nl=4, nc=5, i;
5     int **x;
6     x = malloc(nl * sizeof(int*));
7     for(i=0; i<nl; i++) {
8         x[i] = malloc(nc * sizeof(int));
9     }
10    x[1][2]=3;
11    for(i=0; i<nl; i++) {
12        free (x[i]);
13    }
14    free (x);
15 }
```

Aloca bloco auxiliar

Aloca linhas

Libera bloco auxiliar

Libera linhas

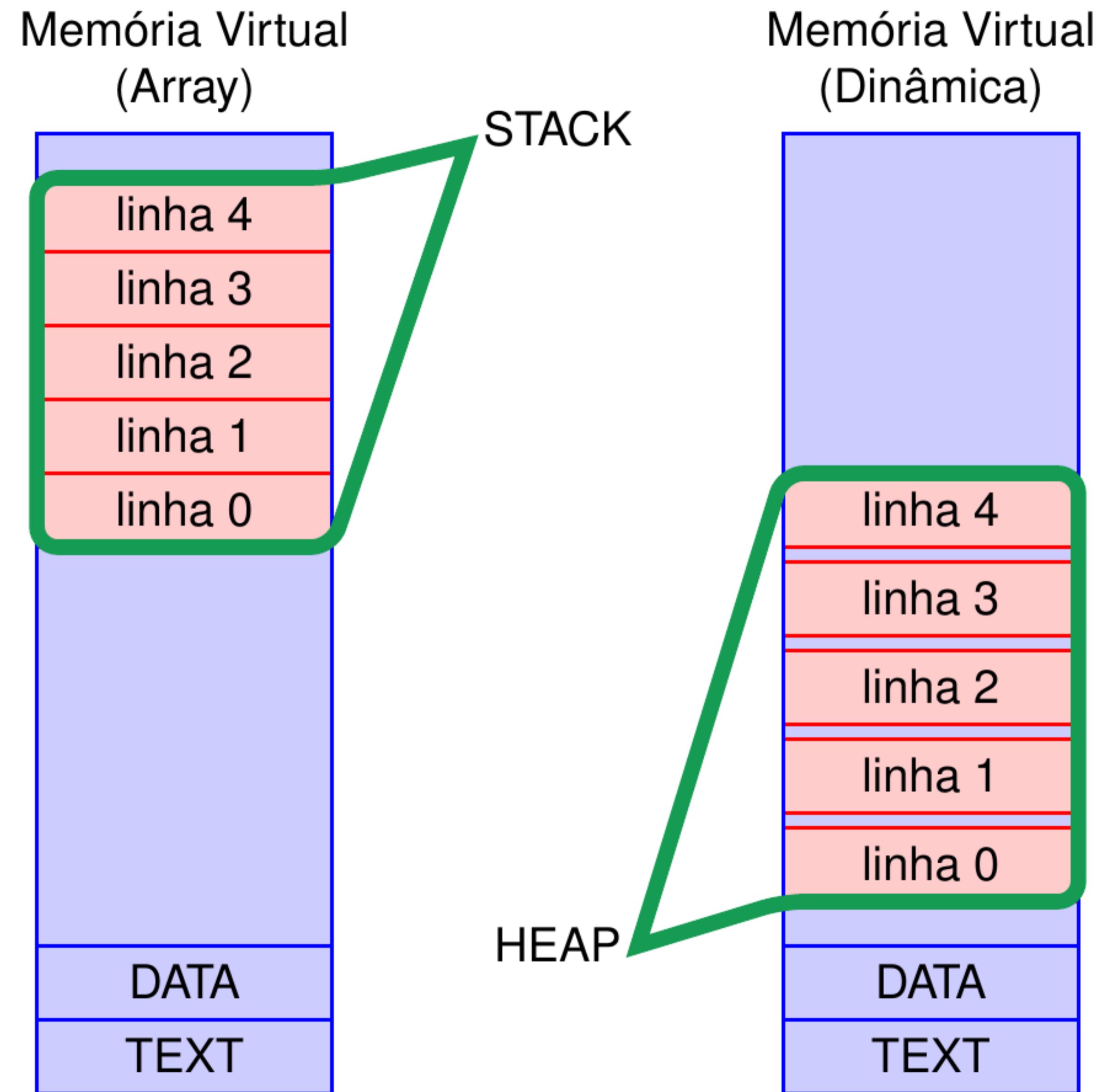


Como se compara esse esquema de alocação realizada com aquela realizada em arrays da forma
`int y[4][5]`?



Explorando a matriz bidimensional...

Criando uma matriz bidimensional (parte I)





Obrigado

Criando uma matriz bidimensional (parte II)

- O esquema de alocação dinâmica da matriz mostrada no exemplo anterior funciona, mas oferece algumas limitações (em tempo e em espaço de memória).
- É mais adequado garantir que os dados da matriz estejam todos agrupados na memória, tanto para facilitar a alocação na memória virtual quanto para realizar operações de cópia.

Solução

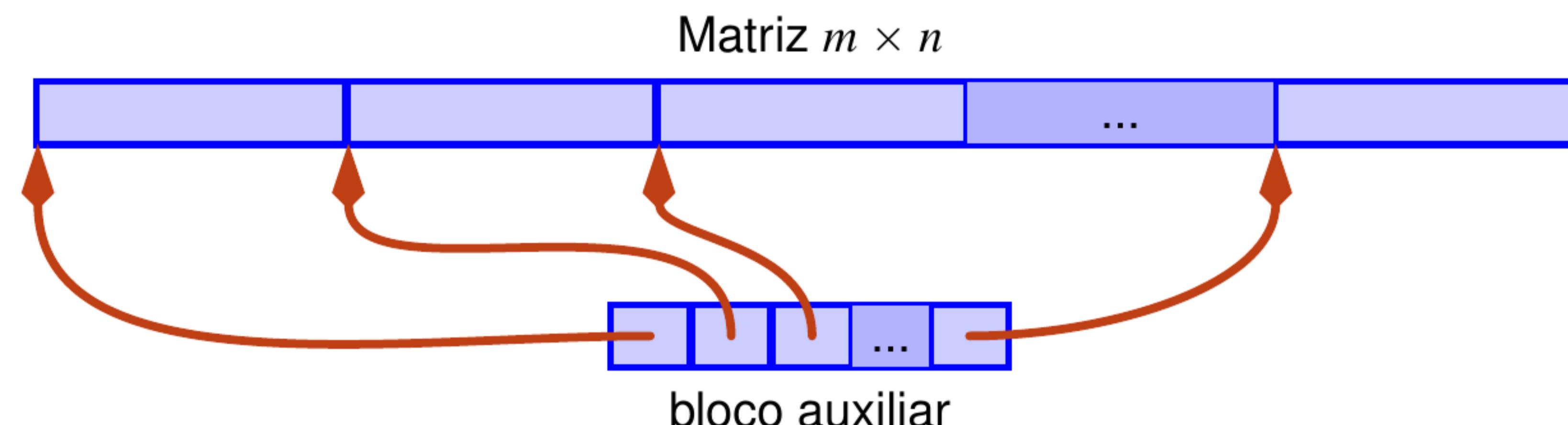
Alocar um único bloco de tamanho $m \times n$ para a matriz e fixar os endereços das linhas em um bloco de memória auxiliar.

Criando uma matriz bidimensional (parte II)

- O esquema de alocação dinâmica da matriz mostrada no exemplo anterior funciona, mas oferece algumas limitações (em tempo e em espaço de memória).
- É mais adequado garantir que os dados da matriz estejam todos agrupados na memória, tanto para facilitar a alocação na memória virtual quanto para realizar operações de cópia.

Solução

Alocar um único bloco de tamanho $m \times n$ para a matriz e fixar os endereços das linhas em um bloco de memória auxiliar.



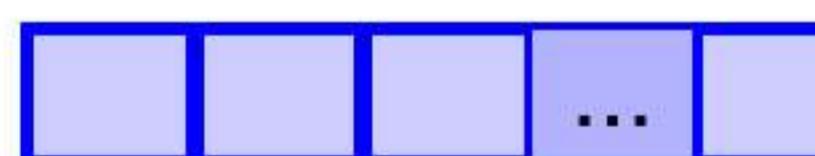
Criando uma matriz bidimensional (parte II)

```
1 int **z;
2 z = malloc(nl*sizeof(int*));
3 z[0] = malloc(nc*nl*sizeof(int));
4 for(i=1; i<nl; i++) {
5     z[i] = z[i-1]+nc;
6 }
7 free(z[0]);
8 free(z);
```

Criando uma matriz bidimensional (parte II)

Aloca bloco auxiliar

```
1 int ***z;
2 z = malloc(nl*sizeof(int*));
3 z[0] = malloc(nc*nl*sizeof(int));
4 for(i=1; i<nl; i++) {
5     z[i] = z[i-1]+nc;
6 }
7 free(z[0]);
8 free(z);
```



bloco auxiliar

Criando uma matriz bidimensional (parte II)

Aloca bloco auxiliar

Aloca linhas

```
1 int **z;
2 z = malloc(nl*sizeof(int*));
3 z[0] = malloc(nc*nl*sizeof(int));
4 for(i=1; i<nl; i++) {
5     z[i] = z[i-1]+nc;
6 }
7 free(z[0]);
8 free(z);
```

Matriz $m \times n$



Criando uma matriz bidimensional (parte II)

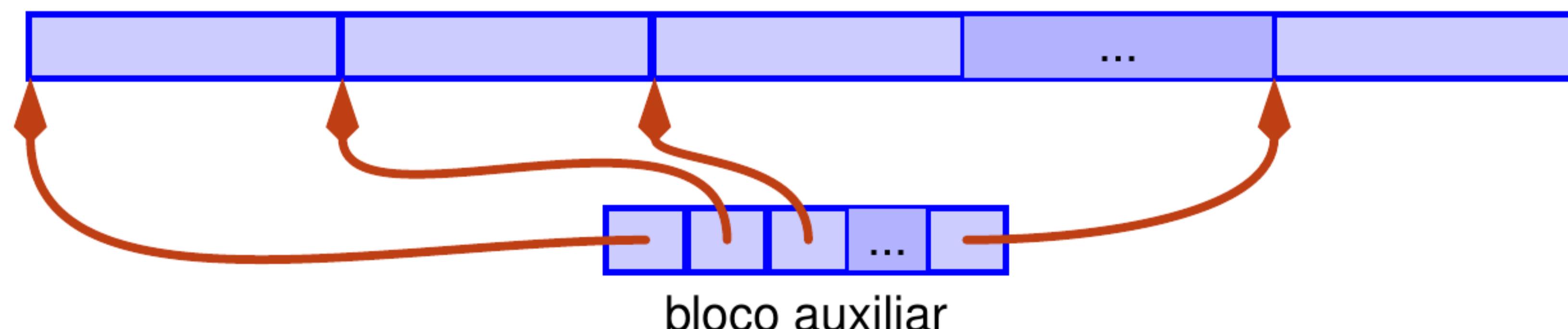
Aloca bloco auxiliar

Aloca linhas

Fixa os ponteiros

```
1 int **z;
2 z = malloc(nl*sizeof(int *));
3 z[0] = malloc(nc*nl*sizeof(int));
4 for(i=1; i<nl; i++) {
5     z[i] = z[i-1]+nc;
6 }
7 free(z[0]);
8 free(z);
```

Matriz $m \times n$



Criando uma matriz bidimensional (parte II)

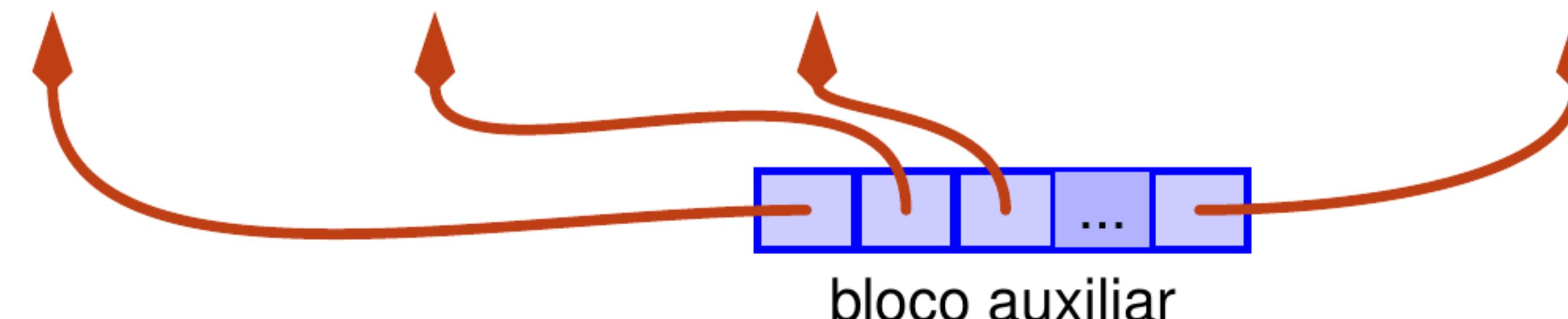
Aloca bloco auxiliar

Aloca linhas

Fixa os ponteiros

```
1 int **z;
2 z = malloc(nl*sizeof(int *));
3 z[0] = malloc(nc*nl*sizeof(int));
4 for(i=1; i<nl; i++) {
5     z[i] = z[i-1]+nc;
6 }
7 free(z[0]);
8 free(z);
```

Libera matriz



Criando uma matriz bidimensional (parte II)

Aloca bloco auxiliar

Aloca linhas

Fixa os ponteiros

```
1 int ***z;
2 z = malloc(nl*sizeof(int*));
3 z[0] = malloc(nc*nl*sizeof(int));
4 for(i=1; i<nl; i++) {
5     z[i] = z[i-1]+nc;
6 }
7 free(z[0]);
8 free(z);
```

Libera auxiliar

Libera matriz



Praticando alocação eficiente...



Obrigado

Ponteiros e funções (parte II)

Retornando ponteiros de funções em C

- Ponteiros também podem ser retornados por funções. A própria função `malloc()` é um bom exemplo disso. Como criar a própria função?

```
int* funcao () {  
}
```

- Exemplo: função que retorna um bloco com `n` valores inteiros aleatórios.

```
1 int* random(int n) {  
2     int *r, i;  
3     r = malloc(n * sizeof(int));  
4     for(i=0; i<n; i++) {  
5         r[i] = rand();  
6     }  
7     return r;  
8 }
```

Retornando ponteiros de funções em C

- Ponteiros também podem ser retornados por funções. A própria função `malloc()` é um bom exemplo disso. Como criar a própria função?

```
int* funcao() {  
}
```

- Exemplo: função que retorna um bloco com `n` valores inteiros aleatórios.

```
1 int* random(int n) {  
2     int *r, i;  
3     r = malloc(n * sizeof(int));  
4     for(i=0; i<n; i++) {  
5         r[i] = rand();  
6     }  
7     return r;  
8 }
```



Praticando o retorno de ponteiros...



Obrigado

Ponteiros e *structs*

Ponteiros e structs

- As estruturas, ou *structs*, são recursos providos pela linguagem C para ajudar o programador a organizar o pensamento.

```
1 struct <identificador> {  
2     <tipo> campo1;  
3     <tipo> campo2;  
4     ...  
5 };
```

```
1 struct cliente{  
2     int id;  
3     char nome[400];  
4     int fone;  
5 };
```

- Permitem que uma ideia (por exemplo, “cliente”) tenha suas características agregadas em uma variável.

```
1 int id[10];  
2 char nome[10][400];  
3 int fone[10];  
4 id[0] = 14394;  
5 fone[0] = 11223344;
```

```
1 struct cliente clientes[10];  
2 clientes[0].id = 14394;  
3 clientes[0].fone=11223344;  
4 printf("id = %d",  
       clientes.id);  
6 printf("fone = %d",  
       clientes.fone);  
7
```

Ponteiros para structs

- Considere o exemplo da estrutura cliente

```
1 struct cliente{  
2     int id;  
3     char nome[400];  
4     int fone;  
5 };  
6 int main(void){  
7     struct cliente jose, *pc;  
8     pc = &jose;  
9     pc->id = 14394;  
10    printf("id = %d\n", pc->id);  
11    printf("digite o fone: ");  
12    scanf("%d", &pc->fone);  
13    printf("fone = %d\n", pc->fone);  
14 }
```

pc->id = 14394;

operador ->

O acesso aos membros da estrutura é feita com o operador ->

pc->id é equivalente a (*pc).id

pc->fone é equivalente a (*pc).fone

Alocação dinâmica de *structs*

- O processo de alocação/liberação de memória é semelhante ao utilizada em variáveis primitivas.
- O único cuidado reside na sintaxe de acesso aos campos da estrutura.

```
1 struct cliente *pc;
2 pc = malloc(10*sizeof(struct cliente));
3 pc[1].id = 14394;
4 printf("id = %d\n", (pc+1)->id);
5 free(pc);
```

- A sintaxe **pc[1].id** referencia o segundo elemento do bloco, no campo `id`. Perceba que o operador `->` não foi usado nesta linha.
- A sintaxe **(pc+1)->id** **TAMBÉM** referencia o segundo elemento do bloco. Ambas as construções são **EQUIVALENTES**.



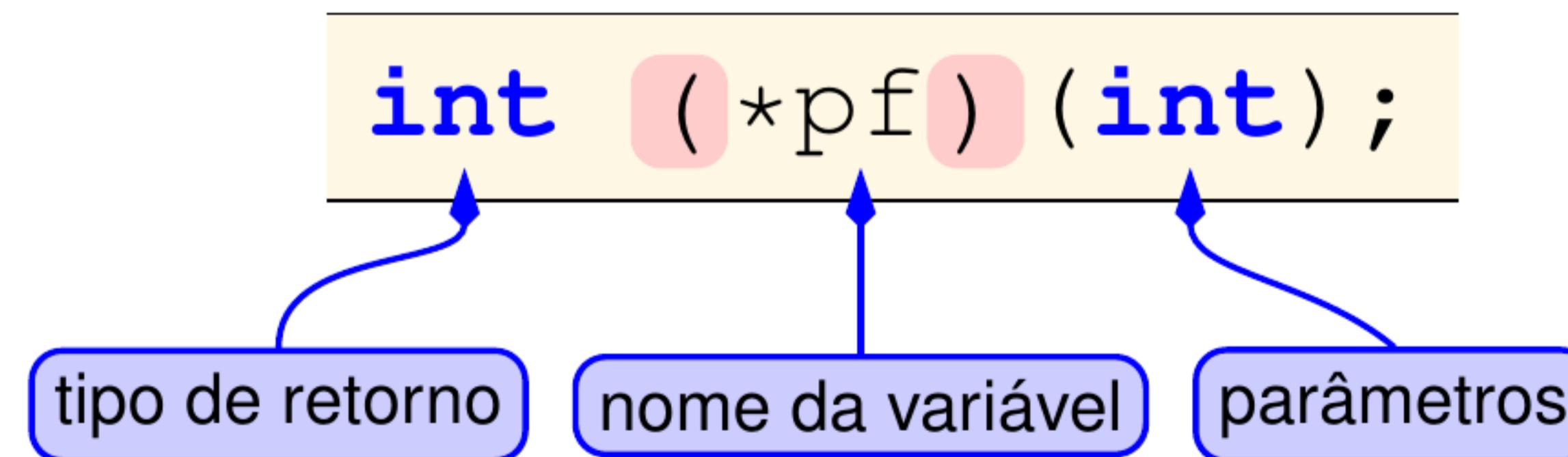
Obrigado

Ponteiros para funções

Ponteiros para funções

- Assim como existem ponteiros para área de dados, existem também ponteiros para funções.
- Ponteiros para funções guardam endereços de áreas de **CÓDIGO**.
- Exemplo de ponteiro para função

```
1 int funcao(int a) {  
2     return a+1;  
3 }  
4 int main(void) {  
5     int (*pf)(int) = &funcao;  
6     printf("valor = %d", pf(1));  
7 }
```



Ponteiros para funções

Muita atenção para os parêntesis ao redor do nome da variável.

Ponteiros para funções

Muita atenção para os parêntesis ao redor do nome da variável.

- Declarado corretamente...

```
int (*pf) (int);
```

... temos o ponteiro para função.

Ponteiros para funções

Muita atenção para os parêntesis ao redor do nome da variável.

- Se remover os parêntesis...

```
int *pf (int);
```

... teríamos a declaração de uma função que recebe um `int` e retorna um ponteiro para `int`.

Ponteiros para funções

Muita atenção para os parêntesis ao redor do nome da variável.

- Se remover os parêntesis...

```
int *pf (int);
```

... teríamos a declaração de uma função que recebe um `int` e retorna um ponteiro para `int`.

- E por falar em remover...

```
int (*pf) (int) = &funcao;
```

Ponteiros para funções

Muita atenção para os parêntesis ao redor do nome da variável.

- Se remover os parêntesis...

```
int *pf (int);
```

... teríamos a declaração de uma função que recebe um `int` e retorna um ponteiro para `int`.

- O `&` pode ser removido da atribuição.

```
int (*pf) (int) = funcao;
```

- O próprio nome da função já funciona para retornar seu endereço.

Arrays de ponteiros para funções

```
1 void soma(int a, int b) {  
2     printf("Soma = %d\n", a+b);  
3 }  
4 void subtracao(int a, int b) {  
5     printf("Subtracao = %d\n", a-b);  
6 }  
7 void multiplicacao(int a, int b) {  
8     printf("Multiplicacao = %d\n", a*b);  
9 }  
10 int main(void) {  
11     void (*pf_array[]) (int, int) = {soma, subtracao, multiplicacao};  
12     int opcao, a = 3, b = 4;  
13     printf("Digite sua escolha: \n");  
14     scanf("%d", &opcao);  
15     (*pf_array[opcao]) (a, b);  
16     return 0;  
17 }
```

- Ponteiros para funções podem também ser passados como argumentos para outras funções.

```
1 int funcao(int a) {  
2     return a+1;  
3 }  
4 int alo(int (*f)(int)) {  
5     printf("ret = %d\n", f(3));  
6 }  
7 int main(void) {  
8     alo(funcao);  
9 }
```

- Essa funcionalidade permite a criação de funções que implementem algoritmos genéricos, que funcionem com quaisquer funções. Exemplos: integração e diferenciação numérica, ordenação, cálculo de mínimos e máximos...

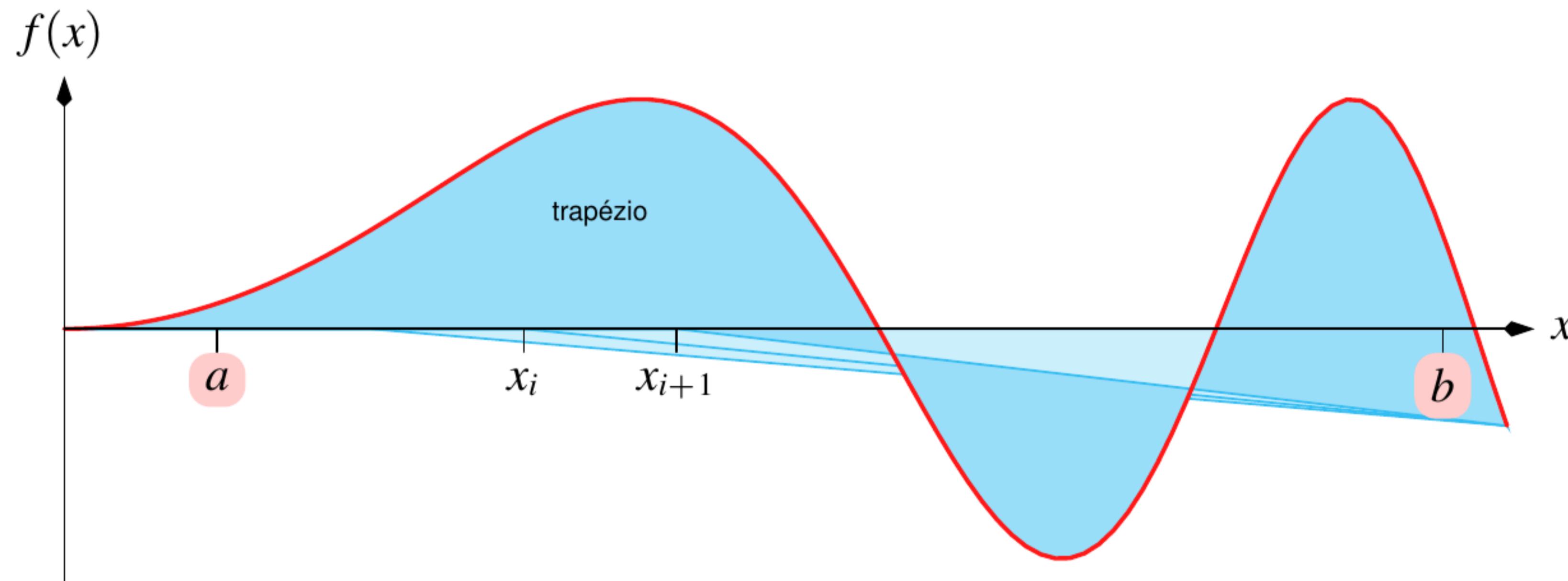


Praticando funções genéricas -

qsort () ...

Criando suas próprias funções genéricas

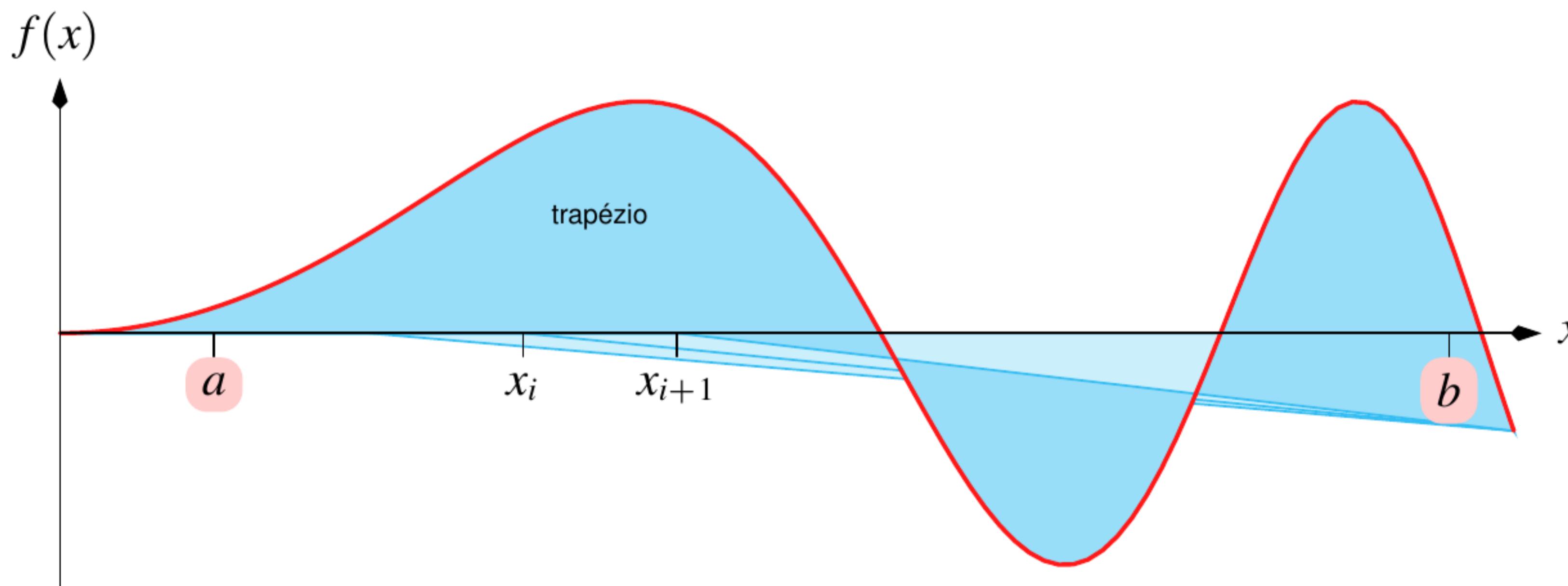
- Exemplo de função genérica: cálculo numéricico da área sob uma curva.



```
float trapezio(float (*f) (float),  
               float a, float b, int n);
```

Criando suas próprias funções genéricas

- Exemplo de função genérica: cálculo numéricico da área sob uma curva.



```
float trapezio(float (*f) (float),  
               float a, float b, int n);
```



Praticando funções genéricas -
trapezio() ...

Considerações finais

Alguns cuidados precisam ser tomados no uso de ponteiros

- **SEMPRE** respeite a regra de alocação/liberação de memória `malloc()` / `free()`.
 - Não deixe ponteiros alocados sem rígido controle para não causar vazamentos de memória.
 - **NUNCA** retorne de uma função ponteiros para arrays locais, salvo se tenham sido declarados como `static`. **Variáveis locais somem** quando suas funções finalizam.
 - É boa prática atribuir o valor `NULL` a ponteiros que não estão armazenando endereços úteis. O valor `NULL` provê uma maneira simples de realizar esse teste.
 - **Não utilize blocos de memória que foram liberados**. Uma vez executada a função `free()` não há mais garantias de que a memória possa ser utilizada.
 - **Não extrapole** o espaço de memória reservado para os blocos alocados no *heap*.

Considerações finais

Alguns cuidados precisam ser tomados no uso de ponteiros

- **SEMPRE** respeite a regra de alocação/liberação de memória `malloc()` / `free()`.
- Não deixe **ponteiros alocados** sem rígido controle para não causar **vazamentos de memória**.
- **NUNCA** retorne de uma função ponteiros para arrays locais, salvo se tenham sido declarados como `static`. **Variáveis locais somem** quando suas funções finalizam.
- É boa prática atribuir o valor **NULL** a ponteiros que não estão armazenando endereços úteis. O valor **NULL** provê uma maneira simples de realizar esse teste.
- **Não utilize blocos de memória que foram liberados**. Uma vez executada a função `free()` não há mais garantias de que a memória possa ser utilizada.
- **Não extrapole** o espaço de memória reservado para os blocos alocados no *heap*.

Considerações finais

Alguns cuidados precisam ser tomados no uso de ponteiros

- **SEMPRE** respeite a regra de alocação/liberação de memória `malloc()` / `free()`.
- Não deixe **ponteiros alocados** sem rígido controle para não causar **vazamentos de memória**.
- **NUNCA** retorne de uma função ponteiros para arrays locais, salvo se tenham sido declarados como `static`. **Variáveis locais somem** quando suas funções finalizam.
- É boa prática atribuir o valor `NULL` a ponteiros que não estão armazenando endereços úteis. O valor `NULL` provê uma maneira simples de realizar esse teste.
- **Não utilize blocos de memória que foram liberados**. Uma vez executada a função `free()` não há mais garantias de que a memória possa ser utilizada.
- **Não extrapole** o espaço de memória reservado para os blocos alocados no *heap*.

Considerações finais

Alguns cuidados precisam ser tomados no uso de ponteiros

- **SEMPRE** respeite a regra de alocação/liberação de memória `malloc()` / `free()`.
- Não deixe **ponteiros alocados** sem rígido controle para não causar **vazamentos de memória**.
- **NUNCA** retorne de uma função ponteiros para arrays locais, salvo se tenham sido declarados como `static`. **Variáveis locais somem** quando suas funções finalizam.
- É boa prática atribuir o valor **NULL** a ponteiros que não estão armazenando endereços úteis. O valor `NULL` provê uma maneira simples de realizar esse teste.
- **Não utilize blocos de memória que foram liberados**. Uma vez executada a função `free()` não há mais garantias de que a memória possa ser utilizada.
- **Não extrapole o espaço de memória** reservado para os blocos alocados no *heap*.

Considerações finais

Alguns cuidados precisam ser tomados no uso de ponteiros

- **SEMPRE** respeite a regra de alocação/liberação de memória `malloc()` / `free()`.
- Não deixe **ponteiros alocados** sem rígido controle para não causar **vazamentos de memória**.
- **NUNCA** retorne de uma função ponteiros para arrays locais, salvo se tenham sido declarados como `static`. **Variáveis locais somem** quando suas funções finalizam.
- É boa prática atribuir o valor **NULL** a ponteiros que não estão armazenando endereços úteis. O valor `NULL` provê uma maneira simples de realizar esse teste.
- **Não utilize blocos de memória que foram liberados**. Uma vez executada a função `free()` não há mais garantias de que a memória possa ser utilizada.
- **Não extrapole o espaço de memória reservado para os blocos alocados no *heap*.**

Considerações finais

Alguns cuidados precisam ser tomados no uso de ponteiros

- **SEMPRE** respeite a regra de alocação/liberação de memória `malloc()` / `free()`.
- Não deixe **ponteiros alocados** sem rígido controle para não causar **vazamentos de memória**.
- **NUNCA** retorne de uma função ponteiros para arrays locais, salvo se tenham sido declarados como `static`. **Variáveis locais somem** quando suas funções finalizam.
- É boa prática atribuir o valor **NULL** a ponteiros que não estão armazenando endereços úteis. O valor `NULL` provê uma maneira simples de realizar esse teste.
- **Não utilize blocos de memória que foram liberados**. Uma vez executada a função `free()` não há mais garantias de que a memória possa ser utilizada.
- **Não extrapole** o espaço de **memória** reservado para os blocos alocados no **heap**.



Obrigado