

Ponteiros (parte 3)

Prof. Paulo Henrique Pisani

outubro/2021

Tópicos

- Ponteiro para ponteiro
- Alocação estática vs Alocação dinâmica (matrizes)
- Passagem de matrizes como parâmetros
- Ponteiro para função

Ponteiro para
ponteiro

Ponteiro para ponteiro

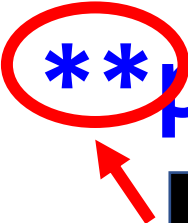
- Ponteiro é uma variável que armazena endereços de memória;
- Um ponteiro para ponteiro armazena o endereço de memória de um ponteiro.

```
int **ptr_i2;  
double **ptr_d2;  
char **ptr_c2;
```

Ponteiro para ponteiro

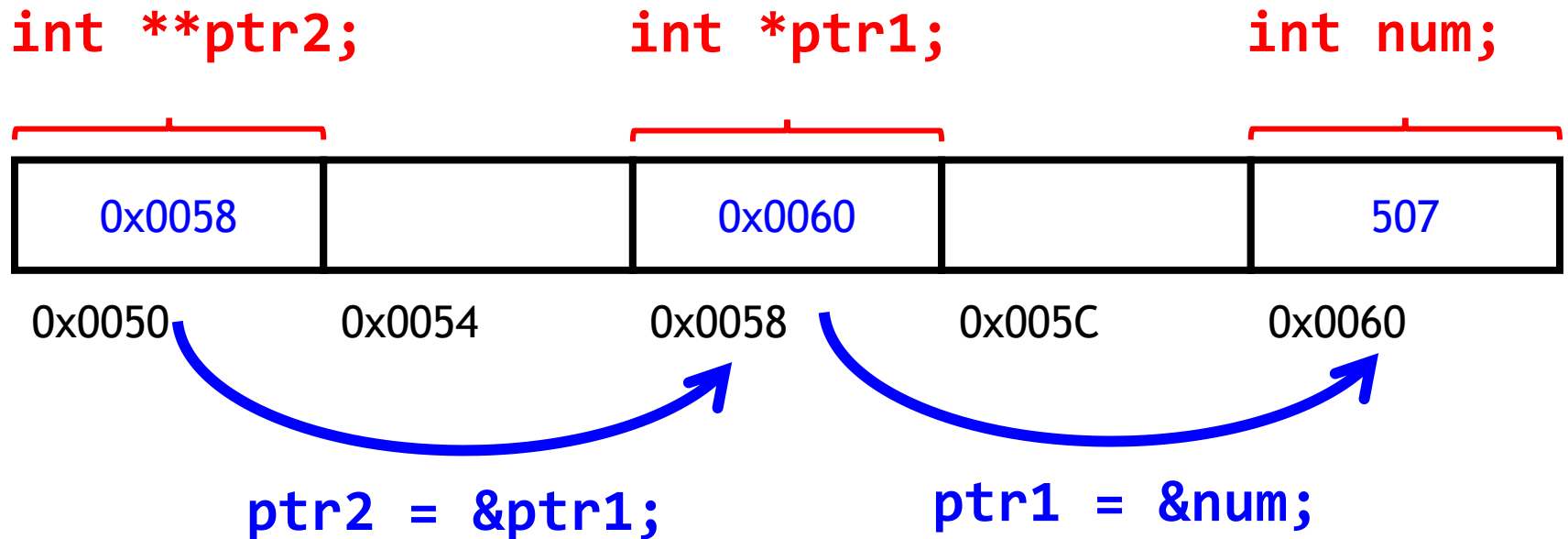
- Ponteiro é uma variável que armazena endereços de memória;
- Um ponteiro para ponteiro armazena o endereço de memória de um ponteiro.

```
int **ptr_i2;  
double **ptr_d2;  
char **ptr_c2;
```



Usamos ** para representar um ponteiro para ponteiro

Ponteiro para ponteiro



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

```
int main() {
```

```
    double nota_p1;
```

```
    scanf("%lf", &nota_p1);  
    printf("%.2lf\n", nota_p1);
```

```
    double *ptr_nota = &nota_p1;
```

```
    scanf("%lf", ptr_nota);  
    printf("%.2lf\n", nota_p1);
```

ptr_nota guarda o endereço de nota_p1

```
    double **ptrptr_nota = &ptr_nota;
```

```
    scanf("%lf", *ptrptr_nota);  
    printf("%.2lf\n", nota_p1);
```

ptrptr_nota aponta para ptr_nota; O “*” retorna o valor da variável que ele aponta, ou seja, o valor de ptr_nota.

```
    return 0;
```

```
}
```

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

```
int main() {
```

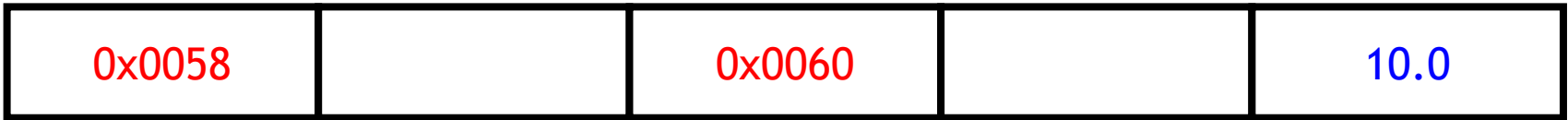
```
double nota_p1;  
scanf("%lf", &nota_p1);  
printf("%.2lf\n", nota_p1);
```

```
double *ptr_nota = &nota_p1;  
scanf("%lf", ptr_nota);  
printf("%.2lf\n", nota_p1);
```

```
double **ptrptr_nota = &ptr_nota;  
scanf("%lf", *ptrptr_nota);  
printf("%.2lf\n", nota_p1);
```

```
return 0;
```

```
}
```



0x0050 0x0054 0x0058 0x005C 0x0060

ptrptr_nota = &ptr_nota;

ptr_nota = ¬a_p1;

Teste 1

```
int x = 2, y = 5;  
int *z = &x;  
int **w = &z;  
**w = y;  
printf("%d\n", x);
```

Teste 2

```
int x = 2, y = 5;  
int *z = &x;  
int **w, **k;  
w = &z;  
*z = 8;  
k = w;  
*k = y;  
printf("%d\n", x);
```

Teste 3

```
int *x, *y;  
x = malloc(sizeof(int));  
y = malloc(sizeof(int));  
int **z = &x;  
int **w, **k;  
*x = 9;  
*y = 11;  
**z = *x + 6;  
w = &y;  
k = w;  
w = z;  
z = k;  
printf("%d %d\n", **z, **w);
```

Alocação dinâmica de parâmetros

- A seguinte função funciona como o esperado?

```
void le_vetor(int *vetor, int n) {  
    vetor = malloc(sizeof(int) * n);  
    int i;  
    for (i = 0; i < n; i++)  
        vetor[i] = rand() % 10;  
}
```

O que é passado por referência e o que é passado por valor?

```
void muda_valor1(int num) {  
    num = 5;  
}
```

```
void muda_valor2(int *num) {  
    *num = 80;  
    num = 5;  
}
```

```
void muda_valor3(int vetor[]) {  
    vetor[0] = 50;  
    vetor = 40;  
}
```

```
void muda_valor4(int *vetor) {  
    vetor[0] = 50;  
    vetor = 30;  
}
```

O que é passado por referência e o que é passado por valor?

```
void muda_valor1(int num) {  
    num = 5;  
}
```



num é passado por valor!

```
void muda_valor2(int *num) {  
    *num = 80;  
    num = 5;  
}
```

```
void muda_valor3(int vetor[]) {  
    vetor[0] = 50;  
    vetor = 40;  
}
```

```
void muda_valor4(int *vetor) {  
    vetor[0] = 50;  
    vetor = 30;  
}
```

O que é passado por referência e o que é passado por valor?

```
void muda_valor1(int num) {  
    num = 5;  
}
```

num é passado por valor!

```
void muda_valor2(int *num) {  
    *num = 80;  
    num = 5;  
}
```

O ponteiro é passado por valor! Mas a variável que o ponteiro aponta é passada por referência!

```
void muda_valor3(int vetor[]) {  
    vetor[0] = 50;  
    vetor = 40;  
}
```

```
void muda_valor4(int *vetor) {  
    vetor[0] = 50;  
    vetor = 30;  
}
```

O que é passado por referência e o que é passado por valor?

```
void muda_valor1(int num) {  
    num = 5;  
}
```

num é passado por valor!

```
void muda_valor2(int *num) {  
    *num = 80;  
    num = 5;  
}
```

O ponteiro é passado por valor! Mas a variável que o ponteiro aponta é passada por referência!

```
void muda_valor3(int vetor[]) {  
    vetor[0] = 50;  
    vetor = 40;  
}
```

```
void muda_valor4(int *vetor) {  
    vetor[0] = 50;  
    vetor = 30;  
}
```

O ponteiro é passado por valor! Mas o vetor que o ponteiro aponta é passado por referência!


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

void le_vetor(int *vetor, int n) {
    vetor = malloc(sizeof(int) * n);
    int i;
    for (i = 0; i < n; i++)
        vetor[i] = rand() % 10;
}

int main() {

    int n=5, *v;
    le_vetor(v, n);

    int i;
    for (i = 0; i < n; i++)
        printf("%d ", v[i]);
    printf("\n");

    free(v);
    return 0;
}
```

O que será
impresso?

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

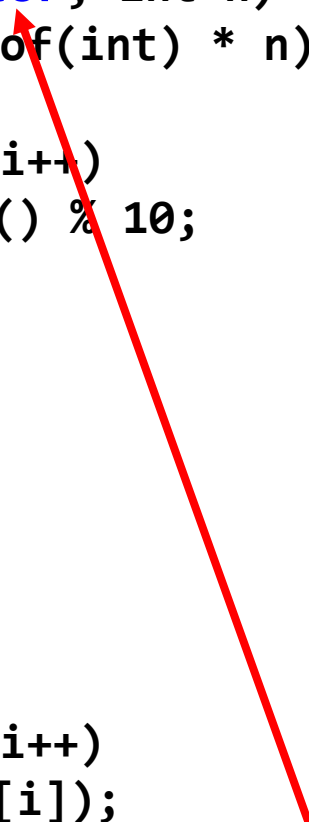
void le_vetor(int *vetor, int n) {
    vetor = malloc(sizeof(int) * n);
    int i;
    for (i = 0; i < n; i++)
        vetor[i] = rand() % 10;
}

int main() {

    int n=5, *v;
    le_vetor(v, n);

    int i;
    for (i = 0; i < n; i++)
        printf("%d ", v[i]);
    printf("\n");

    free(v);
    return 0;
}
```

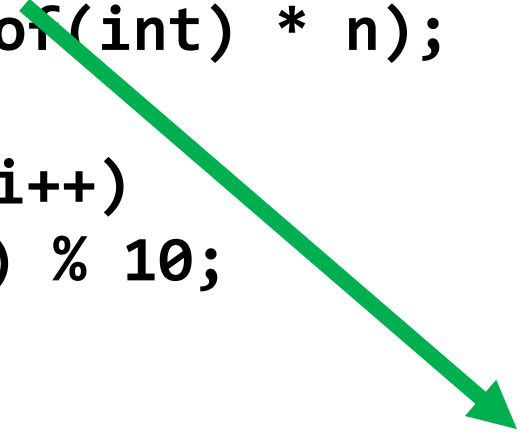


O que será
impresso?

O ponteiro é uma variável e é
passado por valor! Portanto, le_vetor
não altera o valor do ponteiro!!!

Podemos resolver esse problema com ponteiro duplo

```
void le_vetor(int *vetor, int n) {  
    vetor = malloc(sizeof(int) * n);  
    int i;  
    for (i = 0; i < n; i++)  
        vetor[i] = rand() % 10;  
}
```



```
void le_vetor(int **vetor, int n) {  
    *vetor = malloc(sizeof(int) * n);  
    int i;  
    for (i = 0; i < n; i++)  
        (*vetor)[i] = rand() % 10;  
}
```

O ponteiro é passado por valor aqui! Portanto, o retorno de malloc não é armazenado no ponteiro que foi usado na chamada da função!

```
void le_vetor(int *vetor, int n) {  
    vetor = malloc(sizeof(int) * n);  
    int i;  
    for (i = 0; i < n; i++)  
        vetor[i] = rand() % 10;  
}
```

Agora o ponteiro para ponteiro é passado por valor! Mas o *ponteiro* que o ponteiro para ponteiro aponta é passado por referência!

```
void le_vetor(int **vetor, int n) {  
    *vetor = malloc(sizeof(int) * n);  
    int i;  
    for (i = 0; i < n; i++)  
        (*vetor)[i] = rand() % 10;  
}
```

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

void le_vetor(int **vetor, int n) {
    *vetor = malloc(sizeof(int) * n);
    int i;
    for (i = 0; i < n; i++)
        (*vetor)[i] = rand() % 10;
}

int main() {

    int n=5, *v;
    le_vetor(&v, n);

    int i;
    for (i = 0; i < n; i++)
        printf("%d ", v[i]);
    printf("\n");

    free(v);
    return 0;
}
```

O que será
impresso?

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

void le_vetor(int **vetor, int n) {
    *vetor = malloc(sizeof(int) * n);
    int i;
    for (i = 0; i < n; i++)
        (*vetor)[i] = rand() % 10;
}

int main() {

    int n=5, *v;
    le_vetor(&v, n);

    int i;
    for (i = 0; i < n; i++)
        printf("%d ", v[i]);
    printf("\n");

    free(v);
    return 0;
}
```

O que será
impresso?

1 7 4 0 9

Alocação estática vs Alocação dinâmica

(matrizes)

Alocação dinâmica

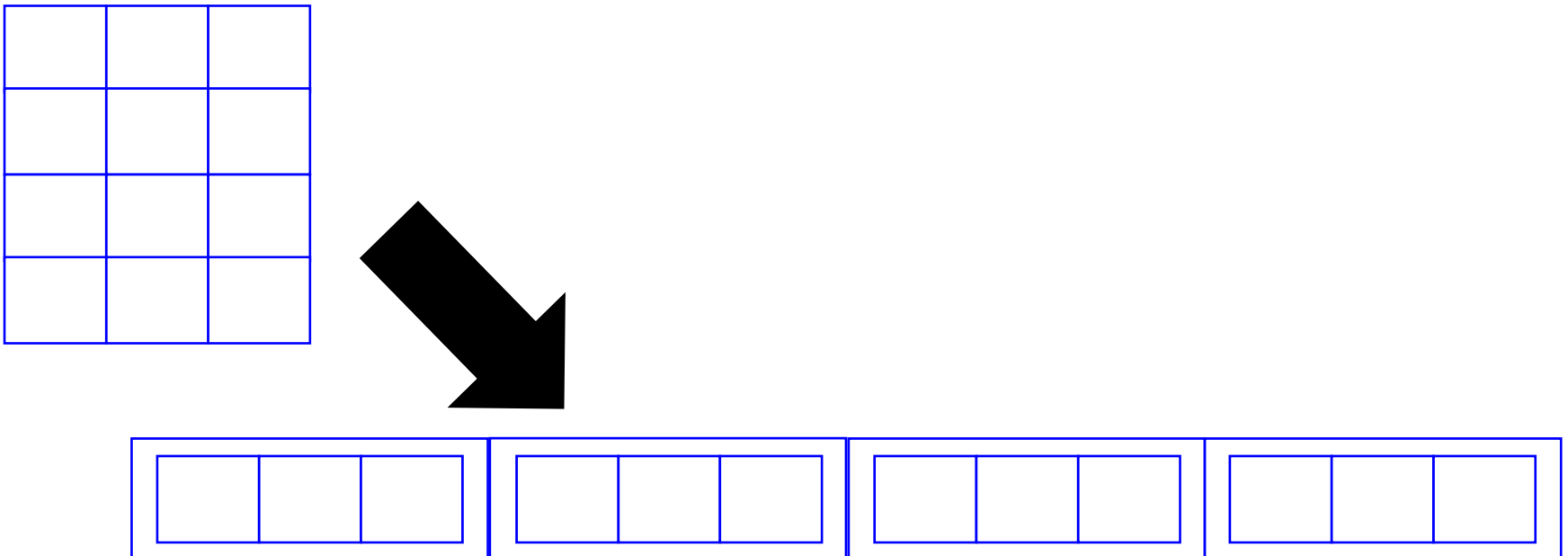
- Vimos que podemos usar malloc e calloc para alocar memória dinamicamente;
- Usamos essas funções para alocar variáveis simples e vetores;

```
int *nota_p2 = malloc(sizeof(int));  
double *notas_alunos = malloc(sizeof(double) * 10);
```

- Agora vamos alocar matrizes dinamicamente!

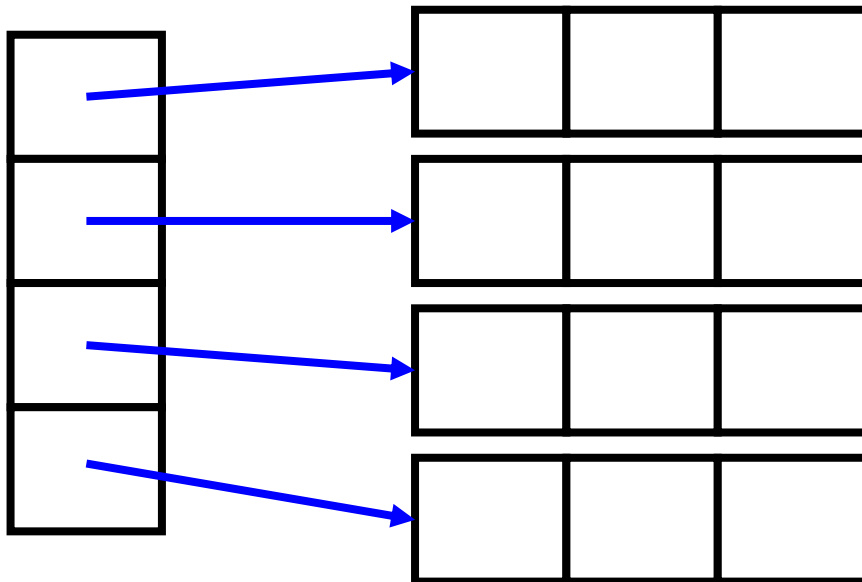
Alocação dinâmica de matrizes

- Uma matriz é um vetor de vetores;
- Portanto, podemos entender uma matriz da seguinte forma:



Alocação dinâmica de matrizes

- Uma matriz é um vetor de vetores;
- Cada vetor é identificado pelo seu ponteiro (endereço do primeiro elemento):
 - Portanto, uma matriz pode ser representada por um vetor de ponteiros.



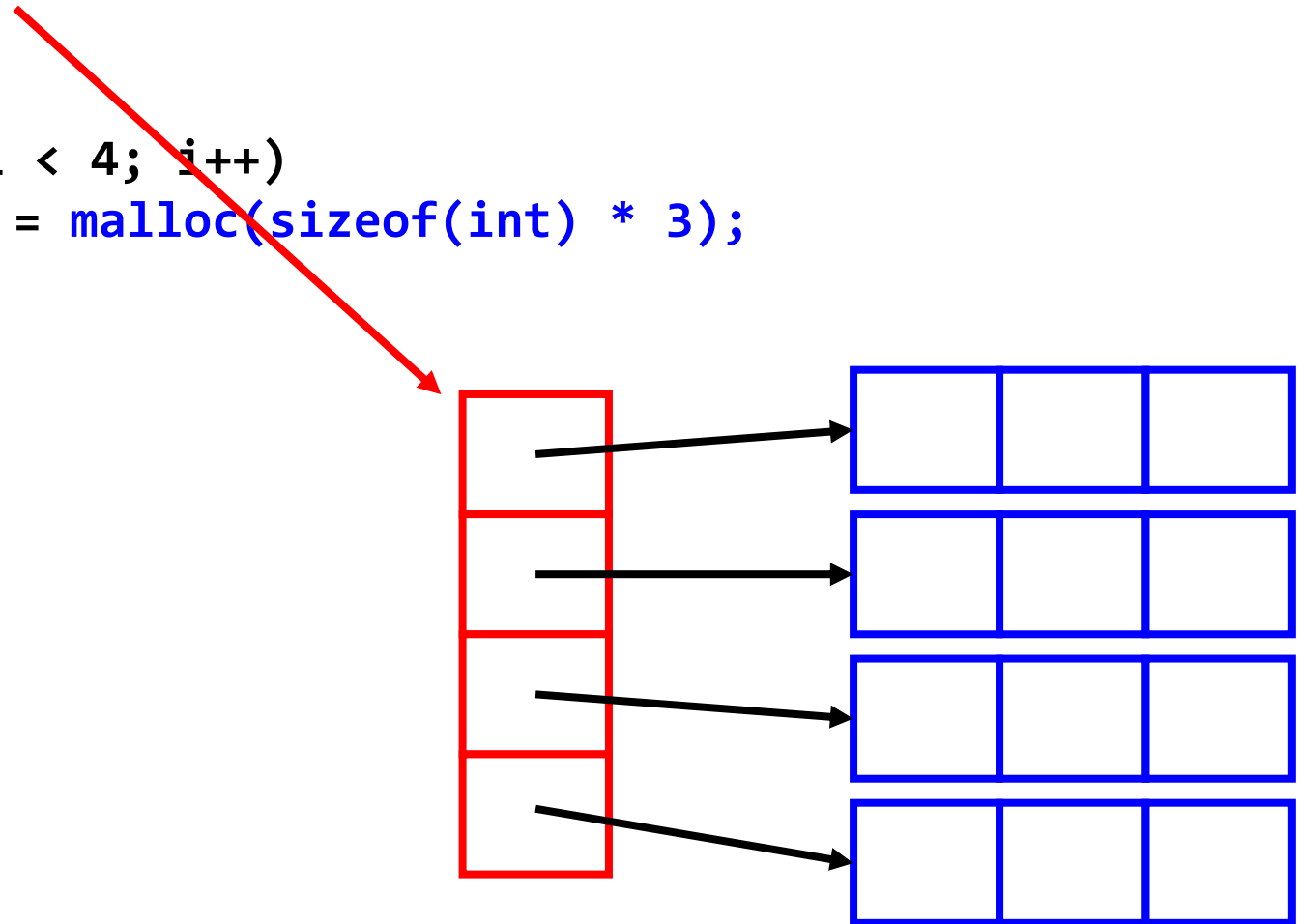
Alocação dinâmica de matrizes

```
int *matriz[4];
```

```
int i;
```

```
for (i = 0; i < 4; i++)
```

```
    matriz[i] = malloc(sizeof(int) * 3);
```



Alocação dinâmica de matrizes

```
/*int *matriz[4];*/
```

```
int **matriz;
```

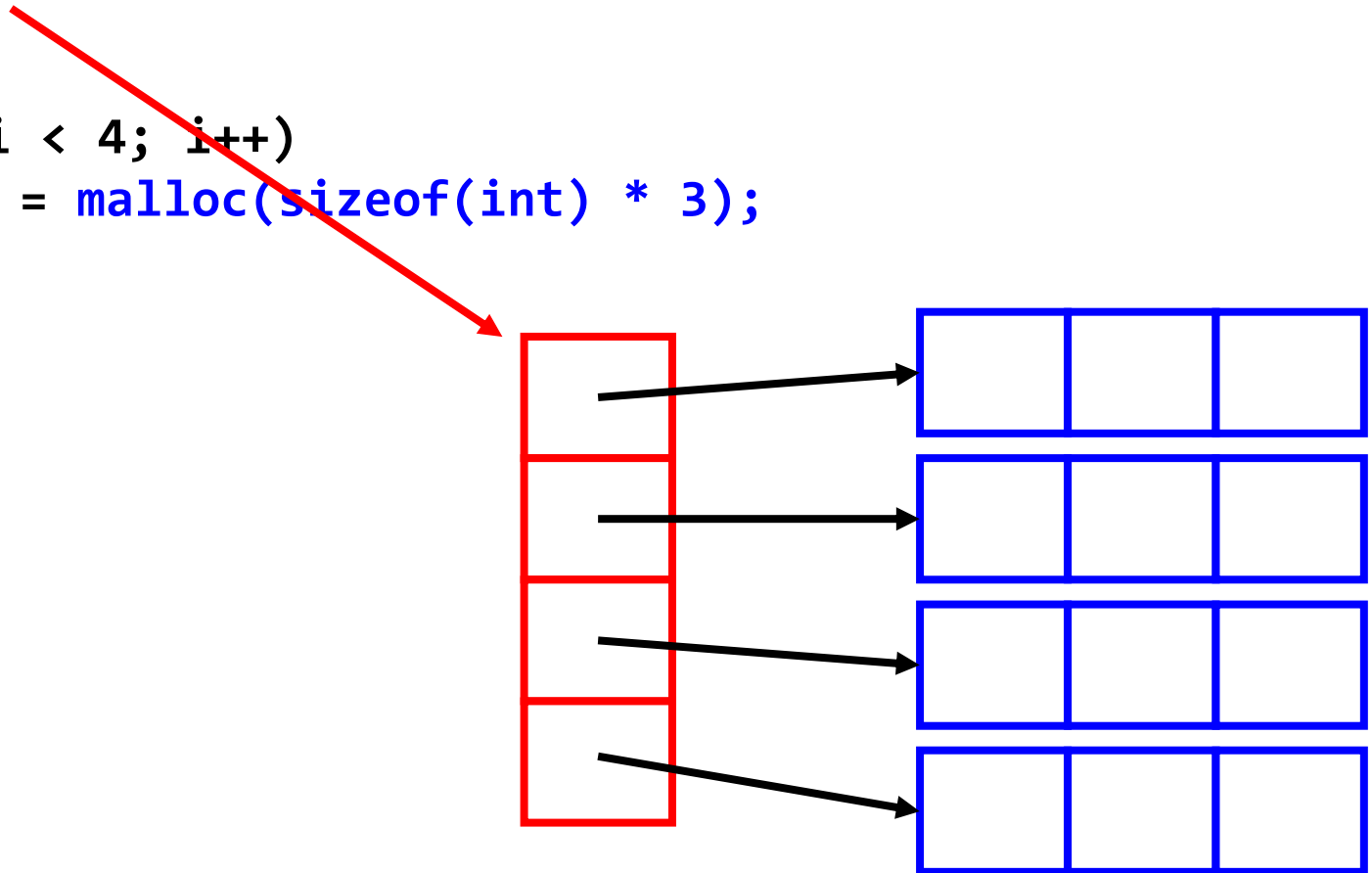
```
matriz = malloc(sizeof(int *) * 4);
```

Para alocar um vetor de ponteiros dinamicamente, usamos um ponteiro para ponteiro.

```
int i;
```

```
for (i = 0; i < 4; i++)
```

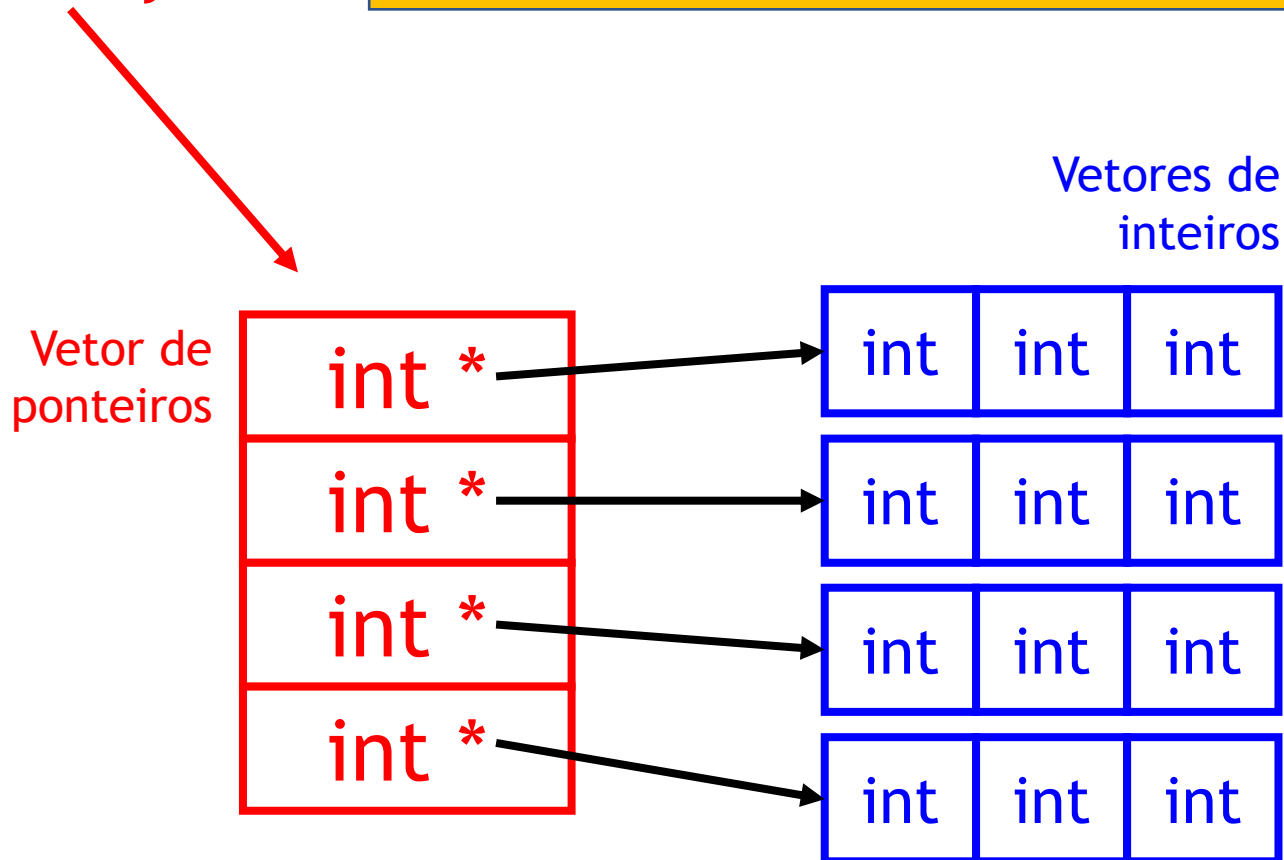
```
    matriz[i] = malloc(sizeof(int) * 3);
```



Alocação dinâmica de matrizes

```
/*int *matriz[4];*/  
int **matriz;
```

Para alocar um vetor de ponteiros dinamicamente, usamos um ponteiro para ponteiro.



Exemplo

- Escreva um programa para alocar uma matriz dinamicamente (usando malloc) e preencher alguns valores em todas posições.

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

int main() {
    int i, j;

    int linhas = 4;
    int colunas = 3;

    // vetor de ponteiros para as linhas
    // observe o sizeof(int*) --> ou seja, o tamanho de um ponteiro de int
    int **matriz = malloc(sizeof(int*) * linhas);

    // para cada ponteiro de linha, alocamos a linha inteira
    for (i = 0; i < linhas; i++) {
        matriz[i] = malloc(sizeof(int) * colunas);
    }

    // Agora a matriz está pronta e podemos colocar alguns valores
    for (i = 0; i < linhas; i++)
        for (j = 0; j < colunas; j++) {
            matriz[i][j] = (i+1)*(j+1);
        }

    // E depois podemos imprimir
    for (i = 0; i < linhas; i++) {
        for (j = 0; j < colunas; j++) {
            printf("%d ", matriz[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

Exercício

- Escreva uma função que retorne uma matriz quadrada com dimensões $n \times n$. A matriz deve ser preenchida com números 1 acima da diagonal principal e com zeros nas demais posições.

0	1	1	1	1
0	0	1	1	1
0	0	0	1	1
0	0	0	0	1
0	0	0	0	0

Exemplo

- Escreva uma função que retorne uma matriz quadrada com dimensões $n \times m$. A função deve ser preenchida com número na sequência, conforme mostrado a seguir:

0	1	2
3	4	5
6	7	8

Função que retorna matriz

- Para retornar uma matriz, basta retornar o ponteiro dela (no caso, um ponteiro para ponteiro):

```
int** retorna_matriz(int linhas, int colunas) {  
    int **m;  
    m = malloc(sizeof(int *) * linhas);  
  
    int c=0;  
    int i, j;  
    for (i = 0; i < linhas; i++) {  
        m[i] = malloc(sizeof(int) * colunas);  
        for (j = 0; j < colunas; j++)  
            m[i][j] = c++;  
    }  
  
    return m;  
}
```

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

```
int** retorna_matriz(int linhas, int colunas) {
    int **m;
    m = malloc(sizeof(int *) * linhas);
    int c=0;
    int i, j;
    for (i = 0; i < linhas; i++) {
        m[i] = malloc(sizeof(int) * colunas);
        for (j = 0; j < colunas; j++)
            m[i][j] = c++;
    }
    return m;
}
```

```
int main() {
    int **matriz;
    matriz = retorna_matriz(4, 3);

    int i, j;
    for (i = 0; i < 4; i++) {
        for (j = 0; j < 3; j++)
            printf("%d ", matriz[i][j]);
        printf("\n");
    }
```

```
// Código para liberar matriz da memória aqui
```

```
return 0;
```

```
}
```

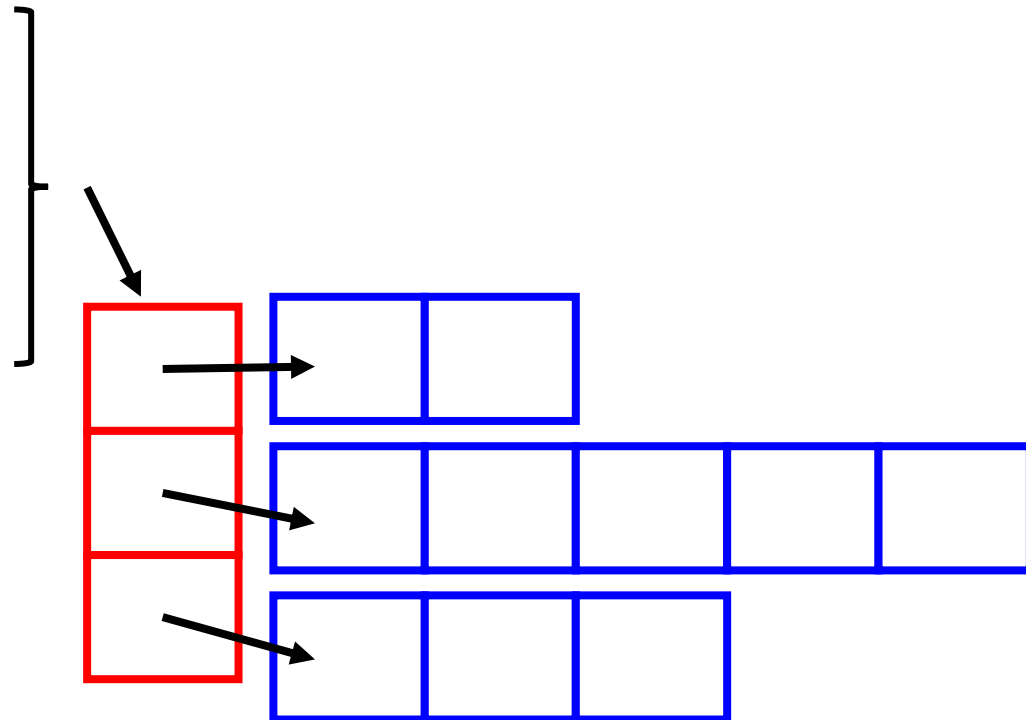
Como implementarr
essa parte do código?



Alocação dinâmica de matrizes

- Utilizando esse mecanismo de alocação dinâmica, podemos ter estruturas em que cada linha tem um comprimento diferente

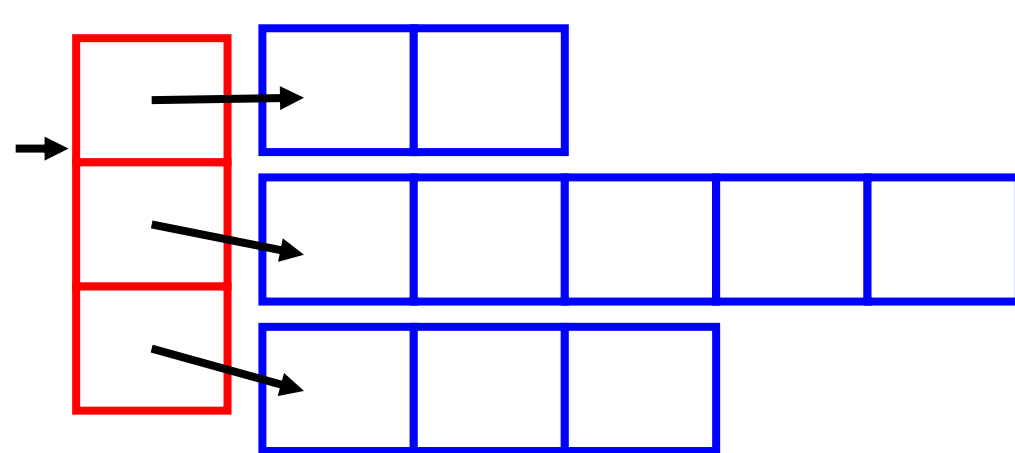
```
int **m;  
m = malloc(sizeof(int *) * 3);  
  
m[0] = malloc(sizeof(int) * 2);  
m[1] = malloc(sizeof(int) * 5);  
m[2] = malloc(sizeof(int) * 3);
```



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

```
int main() {
    int **m;
    m = malloc(sizeof(int *) * 3);

    m[0] = malloc(sizeof(int) * 2);
    m[1] = malloc(sizeof(int) * 5);
    m[2] = malloc(sizeof(int) * 3);
```



```
    int j, c=1;
    for (j = 0; j < 2; j++) {
        m[0][j] = c++;
        printf("%d ", m[0][j]);
    }
    printf("\n");
    for (j = 0; j < 5; j++) {
        m[1][j] = c++;
        printf("%d ", m[1][j]);
    }
    printf("\n");
    for (j = 0; j < 3; j++) {
        m[2][j] = c++;
        printf("%d ", m[2][j]);
    }
    printf("\n");

    free(m[0]); free(m[1]); free(m[2]);
    free(m);

    return 0;
```

```
}
```

Outra alternativa

- Podemos criar uma matriz usando um ponteiro simples também;
- Nesse caso, não usamos a notação com colchetes.

```
int *matriz;
```

```
matriz = malloc(sizeof(int) * 3 * 4);
```

```
int i, j, c=0;  
for (i = 0; i < 4; i++)  
    for (j = 0; j < 3; j++)  
        *(matriz + i * 3 + j) = c++;
```

```

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

int main() {
    int *matriz;

    matriz = malloc(sizeof(int) * 3 * 4);

    int i, j, c=0;
    for (i = 0; i < 4; i++)
        for (j = 0; j < 3; j++)
            *(matriz + i * 3 + j) = c++;

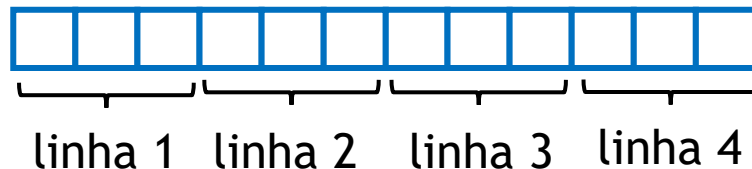
    for (i = 0; i < 4; i++) {
        for (j = 0; j < 3; j++)
            printf("%d ", *(matriz + i * 3 + j));
        printf("\n");
    }

    free(matriz);

    return 0;
}

```

Dessa forma, apenas um vetor é alocado, mas ele é tratado como uma matriz:



**E se quisermos alocar uma matriz em
um parâmetro de função?**

Ponteiro para ponteiro para ponteiro



```
void gera_matriz(int ***matriz_param, int linhas, int colunas) {  
    int **m;  
    m = malloc(sizeof(int *) * linhas);  
  
    int c=0;  
    int i, j;  
    for (i = 0; i < linhas; i++) {  
        m[i] = malloc(sizeof(int) * colunas);  
        for (j = 0; j < 3; j++)  
            m[i][j] = c++;  
    }  
  
    *matriz_param = m;  
}
```

Atenção com o uso de múltiplos níveis de ponteiros! O código pode tornar-se muito difícil de ler!

Ponteiro para função

Ponteiro para função

- Em C, podemos ter ponteiros para funções também!

```
<tipo_retorno> (*nome_ponteiro)(<tipos_parametros>);
```

Ponteiro para função

- Em C, podemos ter ponteiros para funções também!

```
void (*ptr_fun)();  
ptr_fun = &funcao_a;  
  
void (*ptr_fun2)(int);  
ptr_fun2 = &funcao_b;
```

```
void funcao_a() {  
    printf("Estou na funcao A\n");  
}  
  
void funcao_b(int param) {  
    printf("Estou na funcao B\n");  
}
```

Ponteiro para função

- Em C, podemos ter ponteiros para funções também!

```
void (*ptr_fun)();  
ptr_fun = funcao_a;  
  
void (*ptr_fun2)(int);  
ptr_fun2 = &funcao_b;
```

```
void funcao_a() {  
    printf("Estou na funcao A\n");  
}  
  
void funcao_b(int param) {  
    printf("Estou na funcao B\n");  
}
```

No caso de ponteiros de função, não é necessário usar o “&” para obter o endereço de uma função.

```
#include <stdio.h>

double soma(double a, double b) {
    return a + b;
}

double multiplicacao(double a, double b) {
    return a * b;
}

int main() {
    double (*funcao)(double, double);

    funcao = &soma;
    printf("%.2lf\n", funcao(5,6));

    funcao = &multiplicacao;
    printf("%.2lf\n", funcao(5,6));

    return 0;
}
```

O que será
impresso?

```
#include <stdio.h>

double soma(double a, double b) {
    return a + b;
}

double multiplicacao(double a, double b) {
    return a * b;
}

int main() {
    double (*funcao)(double, double);

    funcao = &soma;
    printf("%.2lf\n", funcao(5,6));

    funcao = &multiplicacao;
    printf("%.2lf\n", funcao(5,6));

    return 0;
}
```

O que será
impresso?

11.00
30.00

Vetor de ponteiros de função

- Com um vetor de funções, podemos substituir um switch-case:

```
int operacao;  
scanf("%d", &operacao);  
switch (operacao) {  
    case 0: a = soma(a, b); break;  
    case 1: a = multiplicacao(a, b); break;  
    case 2: a = divisao(a, b); break;  
}
```



```
int operacao;  
scanf("%d", &operacao);  
double(*funcoes[])(double,double) = {&soma, &multiplicacao, &divisao};  
a = funcoes[operacao](a, b);
```



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

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

```
double multiplicacao(double a, double b) {  
    return a * b;  
}
```

```
double divisao(double a, double b) {  
    return a / b;  
}
```

```
int main() {  
    double (*funcoes[])(double, double) = {&soma, &multiplicacao, &divisao};
```

```
    double a = 1, b = 4;
```

```
    int operacao = 0;  
    a = funcoes[operacao](a, b);  
    printf("%.2lf\n", a);
```

```
    operacao = 2;  
    a = funcoes[operacao](a, b);  
    printf("%.2lf\n", a);
```

```
    return 0;
```

```
}
```

O que será
impresso?

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

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

```
double multiplicacao(double a, double b) {  
    return a * b;  
}
```

```
double divisao(double a, double b) {  
    return a / b;  
}
```

```
int main() {  
    double (*funcoes[])(double, double) = {&soma, &multiplicacao, &divisao};
```

```
    double a = 1, b = 4;
```

```
    int operacao = 0;  
    a = funcoes[operacao](a, b);  
    printf("%.2lf\n", a);
```

```
    operacao = 2;  
    a = funcoes[operacao](a, b);  
    printf("%.2lf\n", a);
```

```
    return 0;
```

```
}
```

O que será
impresso?

5.00
1.250

Exercício 1

- Implemente a função “aloca_int” que aloca um inteiro:

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

void aloca_int(int **num) {
    // Implemente funcao aqui
}

int main() {

    int *num;
    aloca_int(&num);

    scanf("%d", num);

    printf("%d\n", *num);

    return 0;
}
```

Exercício 2

- Implemente a função “aloca_vetor” que aloca um vetor e atribui valores aleatórios em todas as posições (use rand() para gerar os números aleatórios).

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

void aloca_vetor(int **v, int n) {
    // Implemente função aqui
}

int main() {

    int n = 10;
    int *v;
    aloca_vetor(&v, n);

    int i;
    for (i = 0; i < n; i++)
        printf("%d ", v[i]);
    printf("\n");

    return 0;
}
```

Exercício 3

- Crie uma função que recebe duas matrizes e retorna o resultado da multiplicação com **“return”**;
- Faça duas implementações:
 - uma que retorna ponteiro simples;
 - outra que retorna ponteiro duplo.

Bibliografia básica

- CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Algoritmos: teoria e prática. 2. ed. Rio de Janeiro, RJ: Campus, 2002.
- FORBELLONE, A. L. V.; EBERSPACHER, H. F. Lógica de programação: a construção de algoritmos e estruturas de dados. 3. ed. São Paulo, SP: Prentice Hall, 2005.
- PINHEIRO, F. A. C. Elementos de programação em C. Porto Alegre, RS: Bookman, 2012.

Bibliografia complementar

- AGUILAR, L. J. Programação em C++: algoritmos, estruturas de dados e objetos. São Paulo, SP: McGraw-Hill, 2008.
- DROZDEK, A. Estrutura de dados e algoritmos em C++. São Paulo, SP: Cengage Learning, 2009.
- KNUTH D. E. The art of computer programming. Upper Saddle River, USA: Addison- Wesley, 2005.
- SEDGEWICK, R. Algorithms in C++: parts 1-4: fundamentals, data structures, sorting, searching. Reading, USA: Addison-Wesley, 1998.
- SZWARCFITER, J. L.; MARKENZON, L. Estruturas de dados e seus algoritmos. 3. ed. Rio de Janeiro, RJ: LTC, 1994.
- TENENBAUM, A. M.; LANGSAM, Y.; AUGENSTEIN, M. J. Estruturas de dados usando C. São Paulo, SP: Pearson Makron Books, 1995.