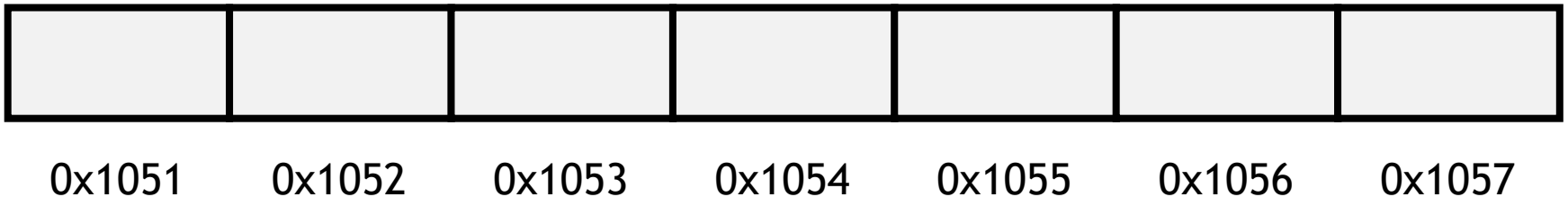
Ponteiros e Alocação de Memória

Prof. Paulo Henrique Pisani

fevereiro/2019

Memória

- Podemos entender a memória como um grande vetor de bytes devidamente endereçados:

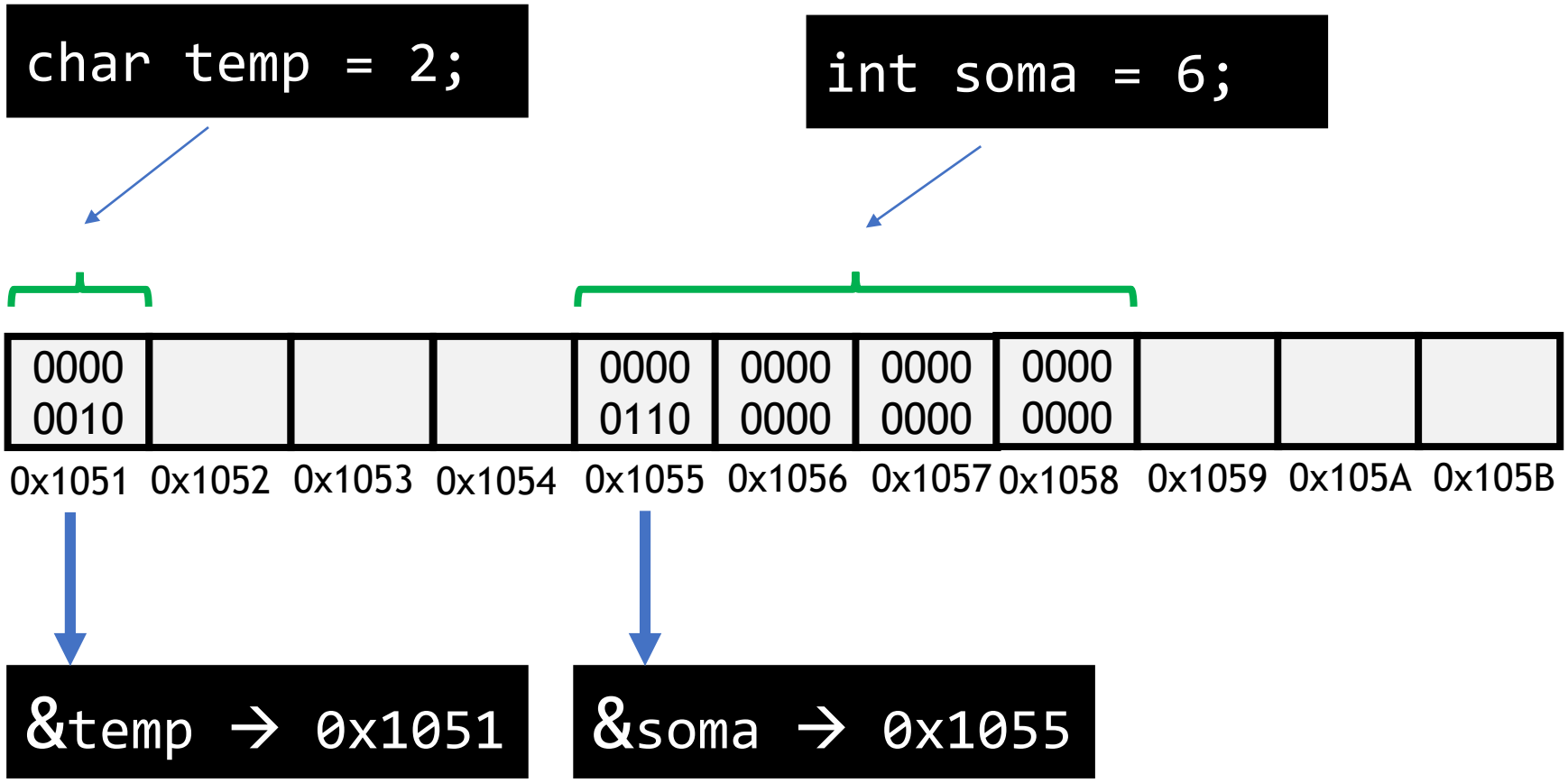


&

Este é o operador address-of!
Ele retorna o endereço do item a sua direita!

Por exemplo:
`&temp` retorna o endereço de `temp`
`&soma` retorna o endereço de `soma`

Endereço de uma variável



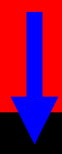
Endereço de uma variável

```
#include <stdio.h>

int main() {
    int num = 800;
    printf("Endereco do num=%d eh %p\n", num, &num);
    return 0;
}
```

```
Endereco do num=800 eh 0060FF0C
```

Ponteiro

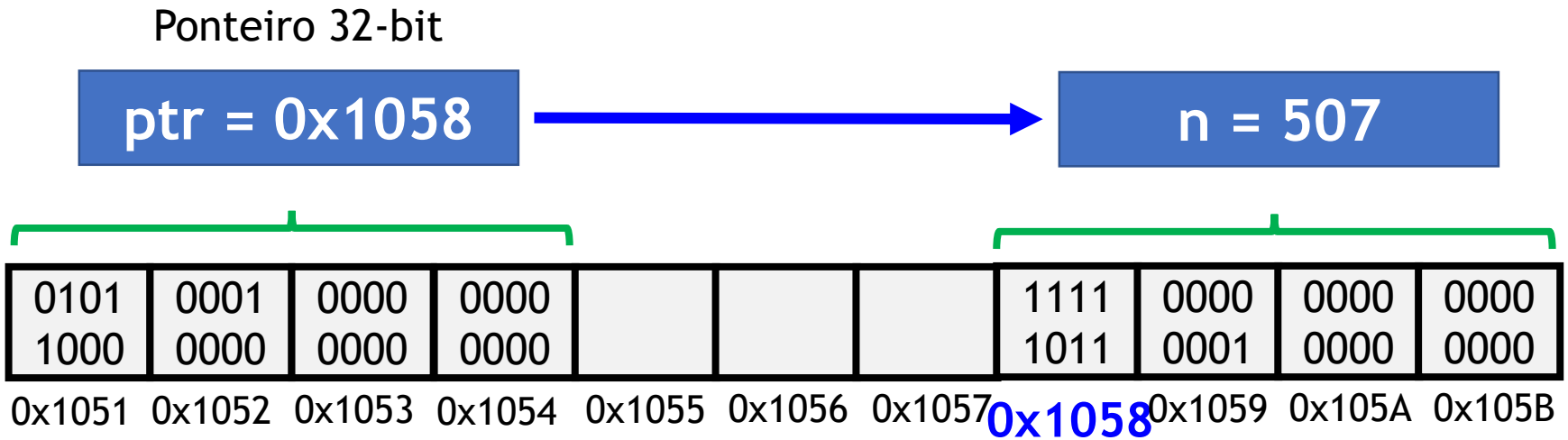
- **Ponteiro é uma variável que armazena um endereço de memória;**
- Para declarar um ponteiro basta incluir um **asterisco** antes do nome da variável:



```
char *ponteiro1;
int *ponteiro2;
double *ponteiro3;
```

Ponteiro

```
int n = 507;
int *ptr;
ptr = &n;
```



Acessando o valor no endereço

```
int n = 507;
int *ptr = &n;
```

Declaração de uma variável do tipo int

Declaração de um ponteiro, que é inicializado apontando para *n*

```
*ptr = 25;
```

Altera o valor da variável que ptr aponta

Acessando o valor no endereço

```
int n = 507;
int *ptr;
ptr = &n;
```

```
*ptr = *ptr + 1;
printf("%d\n", n);
printf("%d\n", *ptr);
```

O que será impresso?

Teste 1

```
int x = 2;
int *y = &x;
*y = 3;
printf("%d\n", x);
```

O que será impresso?

Lista de exercícios do Prof. Fabrício Olivetti:
http://folivetti.github.io/courses/ProgramacaoEstruturada/PDF/exerciciosPonteiros.pdf

Teste 2

```
int x = 10;
int *y = &x;
int *z = &x;
int c = *y + *z;
*y = c;
printf("%d\n", x);
```

O que será impresso?

Lista de exercícios do Prof. Fabrício Olivetti:
http://folivetti.github.io/courses/ProgramacaoEstruturada/PDF/exerciciosPonteiros.pdf

Teste 3

```
int x = 8;
x++;
int *y = &x;
*y = *y + 1;
printf("%d\n", x);
```

O que será impresso?

Lista de exercícios do Prof. Fabrício Olivetti:
http://folivetti.github.io/courses/ProgramacaoEstruturada/PDF/exerciciosPonteiros.pdf

Teste 4

```
int x = 8;
x++;
int *y = &x;
y = y + 1;
printf("%d\n", x);
```

O que será impresso?

Lista de exercícios do Prof. Fabrício Olivetti:
http://folivetti.github.io/courses/ProgramacaoEstruturada/PDF/exerciciosPonteiros.pdf

Teste 5

```
int a = 507;
double b = 300;
int *c;
*c = 30;
*c = &a;
printf("%d ", a);
*c = 10;
printf("%d ", a);
c = &a;
*c = 10;
printf("%d ", a);
```

O que será impresso?

Teste 5

```
int a = 507;
double b = 300;
int *c;
*c = 30;
*c = &a;
printf("%d ", a);
*c = 10;
printf("%d ", a);
c = &a;
*c = 10;
printf("%d ", a);
```

O valor de c não foi inicializado!
Portanto, estamos apontando para uma área indeterminada da memória!

Passagem de parâmetros

Parâmetros são passados por valor

```
#include <stdio.h>

void muda_valor(int parametro) {
    parametro = 507;

    printf("%d\n", parametro);
}

int main() {

    int n = 1000;

    muda_valor(n);

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

    return 0;
}
```

Qual a saída desse programa?

Parâmetros são passados por valor

```
#include <stdio.h>

void muda_valor(int parametro) {
    parametro = 507;

    printf("%d\n", parametro);
}

int main() {

    int n = 1000;

    muda_valor(n);

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

    return 0;
}
```

Ok, variáveis são passadas por valor!

Qual a saída desse programa?

```
507
1000
```

Passagem de parâmetros por referência

“Para passar parâmetros por referência precisamos passar ponteiros por valor”

```c
#include <stdio.h>

void muda_valor_a(double param) {
    param = 99;
    printf("A=%lf\n", param);
}

void muda_valor_b(double *param) {
    *param = 99;
    printf("B=%lf\n", *param);
}

int main() {
    double n = 507;
    printf("%lf\n", n);
    muda_valor_a(n);
    printf("%lf\n", n);
    muda_valor_b(&n);
    printf("%lf\n", n);

    return 0;
}
```

Qual a saída desse programa?

```c
#include <stdio.h>

void muda_valor_a(double param) {
    param = 99;
    printf("A=%lf\n", param);
}

void muda_valor_b(double *param) {
    *param = 99;
    printf("B=%lf\n", *param);
}

int main() {
    double n = 507;
    printf("%lf\n", n);
    muda_valor_a(n);
    printf("%lf\n", n);
    muda_valor_b(&n);
    printf("%lf\n", n);

    return 0;
}
```

Qual a saída desse programa?

```
507.000000
A=99.000000
507.000000
B=99.000000
99.000000
```

scanf

- O **scanf** é uma função que recebe argumentos passados por referência (o valor das variáveis é alterado!);
- Por isso, usamos o operador **&** (*address of*).
  Ou seja, temos que passar o endereço da variável no scanf!

scanf

- Se já temos um endereço de memória, podemos passar ele diretamente (sem usar o &):

```
#include <stdio.h>

int main() {
   int num;
   int *pt1 = &num;

   scanf("%d", pt1);
   printf("%d\n", num);

   return 0;
}
```

Alocação dinâmica

Alocação dinâmica

- Para alocar memória, podemos usar o **malloc**:

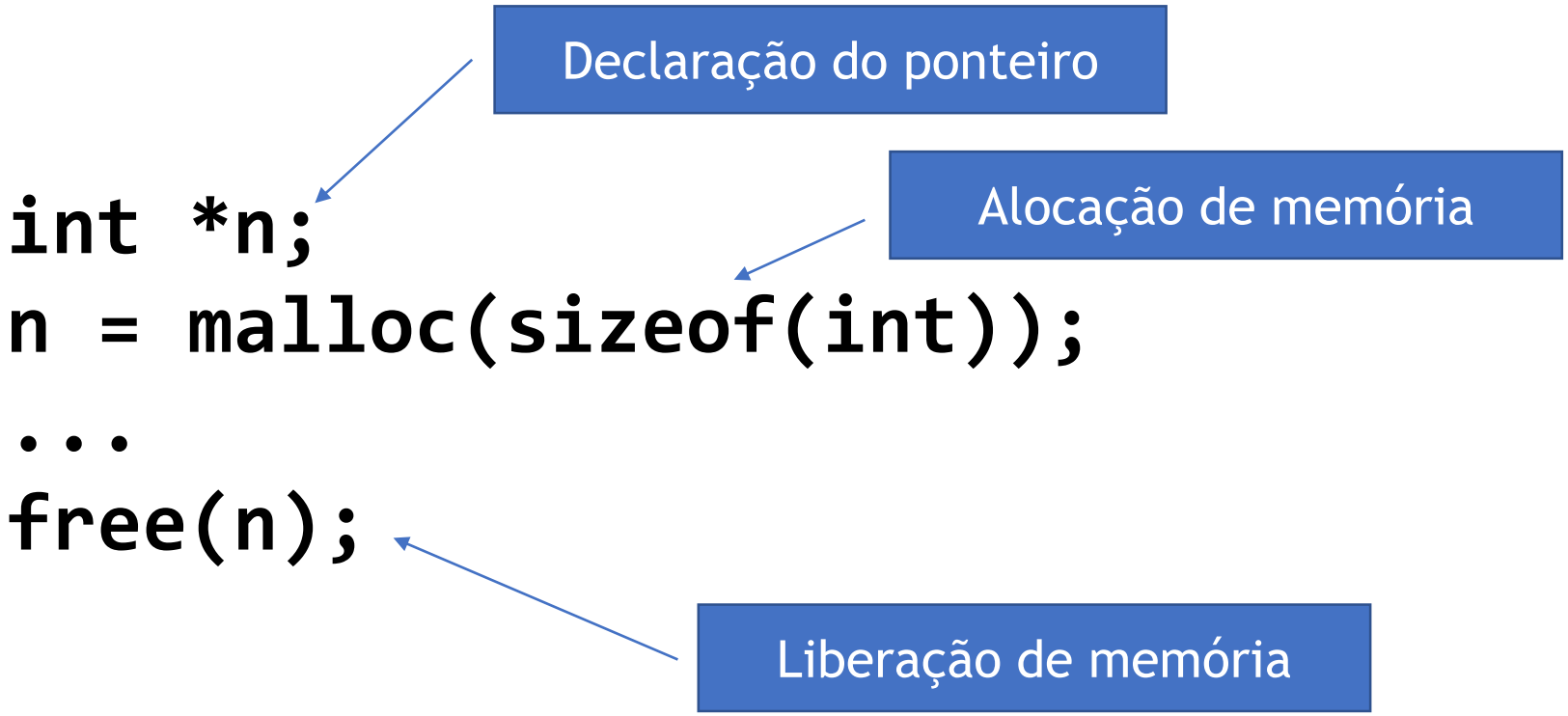
```
void* malloc( size_t size );
```

- Para liberar a memória, usamos o **free**:

```
void free( void* ptr );
```

#include <stdlib.h>

Alocação dinâmica



Alocação dinâmica

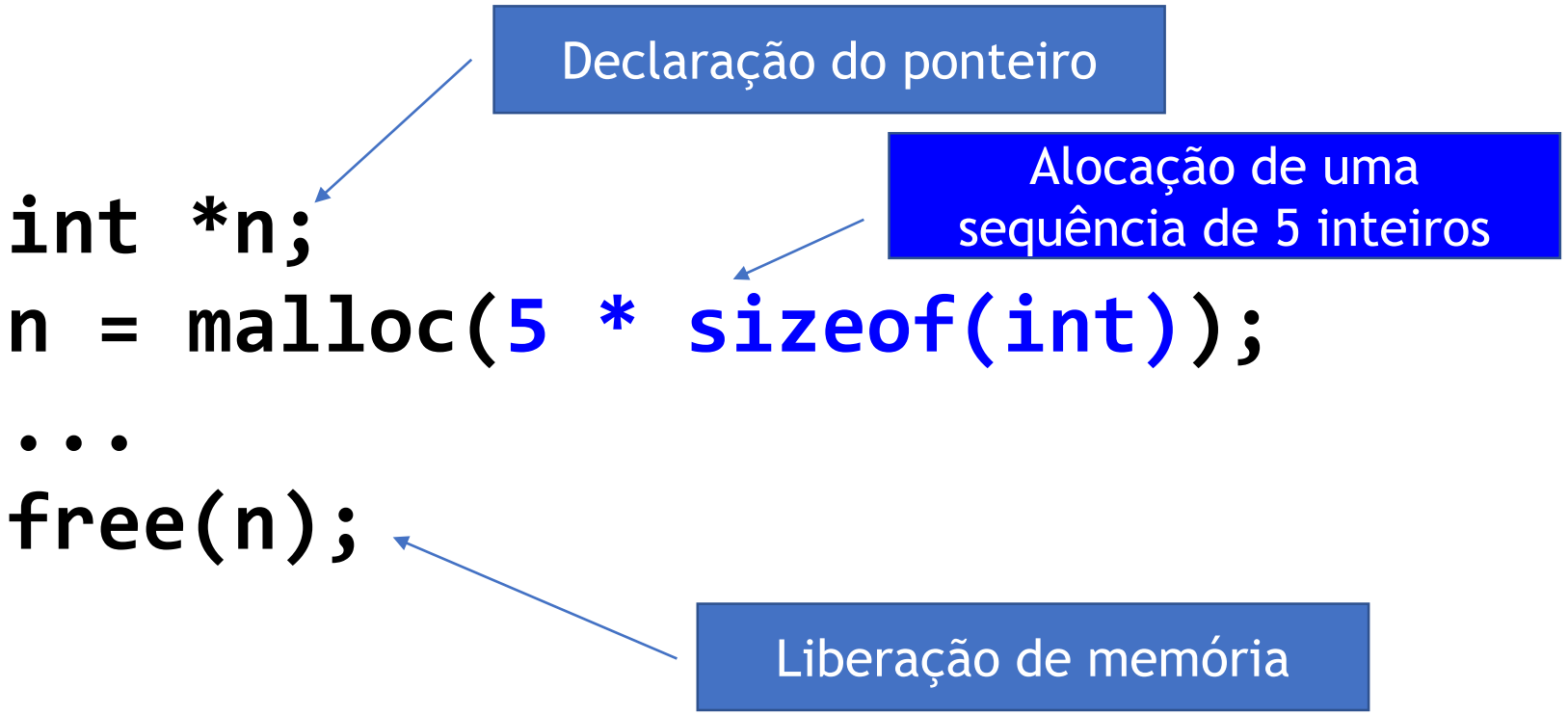
Não use um cast para a saída de malloc

```
int *n;
n = (int *) malloc(sizeof(int));
...
free(n);
```

Cast em malloc

- Caso queira mudar o tipo da variável no código, terá menos trabalho;
- Pode esconder um erro de atribuição de variáveis de tipos distintos;
- Código muito verboso e repetitivo.

Alocação dinâmica (vetores)



Lembre-se de sempre liberar a memória alocada!

```
int *n;
n = malloc(sizeof(int));
...
free(n);
```

```
free(n);
```

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

int main() {
    int n;
    scanf("%d", &n);

    int *vetor = malloc(sizeof(int) * n);
    if (vetor == NULL) {
        printf("Erro na alocacao.\n");
        return -1;
    }
    int i;
    for (i = n-1; i >= 0; i--)
        vetor[i] = n - i;

    free(vetor);

    return 0;
}
```

Importante: Não há garantia que a memória seja alocada! Em caso de erro, é retornado o ponteiro NULL (internamente, é o valor zero)

Alocação dinâmica (vetores)

```c
#include<stdio.h>

int main() {
    int n;
    scanf("%d", &n);

    int vetor[n];

    int i;
    for (i = n-1; i >= 0; i--)
        vetor[i] = n - i;

    return 0;
}
```

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

int main() {
    int n;
    scanf("%d", &n);

    int *vetor = malloc(sizeof(int) * n);

    int i;
    for (i = n-1; i >= 0; i--)
        vetor[i] = n - i;

    free(vetor);

    return 0;
}
```

Aritmética de ponteiros

Aritmética de ponteiros

- Podemos alguns operadores aritméticos sobre ponteiros:
  a) ++
  b) --
  c) +
  d) -

Aritmética de ponteiros

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

int main() {
    int *ptr1 = malloc(sizeof(int));
    char *ptr2 = malloc(sizeof(char));
    double *ptr3 = malloc(sizeof(double));

    printf("%p %p %p\n", ptr1, ptr2, ptr3);

    ptr1 = ptr1 + 3;
    ptr2 = ptr2 + 3;
    ptr3 = ptr3 + 3;

    printf("%p %p %p\n", ptr1, ptr2, ptr3);

    return 0;
}
```

O que será impresso no segundo printf?

Saída

```
00BF0D00 00BF0D20 00BF0D30
?
```

Aritmética de ponteiros

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

int main() {
    int *ptr1 = malloc(sizeof(int));
    char *ptr2 = malloc(sizeof(char));
    double *ptr3 = malloc(sizeof(double));

    printf("%p %p %p\n", ptr1, ptr2, ptr3);

    ptr1 = ptr1 + 3;
    ptr2 = ptr2 + 3;
    ptr3 = ptr3 + 3;

    printf("%p %p %p\n", ptr1, ptr2, ptr3);

    return 0;
}
```

O que será impresso no segundo printf?

Saída

```
00BF0D00 00BF0D20 00BF0D30
00BF0D0C 00BF0D23 00BF0D48
```

Aritmética de ponteiros

- **O efeito das operações aritméticas sobre os ponteiros depende de como foram declarados!**

```
int *ptr1 = malloc(sizeof(int));
char *ptr2 = malloc(sizeof(char));
double *ptr3 = malloc(sizeof(double));
```

```
ptr1 = ptr1 + 1; ⟶ Incrementa 4 (int = 4 bytes)
ptr2 = ptr2 + 1; ⟶ Incrementa 1 (char = 1 byte)
ptr3 = ptr3 + 1; ⟶ Incrementa 8 (double = 8 bytes)
```

Usando aritmética de ponteiros em vetores

- Podemos usar aritmética de vetores para acessar posições de um vetor:

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

```
*(vetor + 0) = 80  <------------>  vetor[0] = 80
*(vetor + 4) = 507 <------------>  vetor[4] = 507
```

Usando aritmética de ponteiros em vetores

- Podemos usar aritmética de vetores para acessar posições de um vetor:

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

```
*(vetor + 0) = 80   <------>   vetor[0] = 80
*(vetor + 4) = 507  <------>   vetor[4] = 507
```

Importante! Coloque parênteses! Assim a aritmética de ponteiros é realizada antes!

Usando aritmética de ponteiros em vetores

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

int main() {
    int *ptr = malloc(sizeof(int) * 10);

    int i;
    for (i = 0; i < 10; i++)
        *(ptr + i) = i*i;

    for (i = 0; i < 10; i++)
        printf("%d ", ptr[i]);
    printf("\n");

    return 0;
}
```

O que será impresso?

Usando aritmética de ponteiros em vetores

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

int main() {
    int *ptr = malloc(sizeof(int) * 10);

    int i;
    for (i = 0; i < 10; i++)
        *(ptr + i) = i*i;

    for (i = 0; i < 10; i++)
        printf("%d ", ptr[i]);
    printf("\n");

    return 0;
}
```

O que será impresso?

0 1 4 9 16 25 36 49 64 81

Vetores como parâmetro e como retorno de função

Passagem de vetor como parâmetro

- Vetores são passados por referência:

```
#include <stdio.h>

void muda_valor(int vetor[]) {
    vetor[0] = 90;

    printf("%d\n", vetor[0]);
}

int main() {
    int v[3] = {200, 500, 300};

    muda_valor(v);

    printf("%d %d %d\n",
        v[0], v[1], v[2]);

    return 0;
}
```

Mas por-que é assim?

Variáveis

```
int matricula = 123;
```

- O identificador de uma váriavel é usado para acessar seu valor;

```
printf("%d\n", matricula);
```

- O endereço de memória da variável é acessado com o operador address-of &

```
printf("%p\n", &matricula);
```

Vetores

```
int vetor[3] = {20, 500, 7};
```

- O identificador de um vetor representa o **endereço do primeiro elemento**!

```
printf("%p\n", vetor);
printf("%p\n", &vetor[0]);
```

Retorna o mesmo valor nos dois casos!

```c
#include <stdio.h>

void muda_valor(int vetor[]) {
    vetor[0] = 90;
    vetor[1] = 507;

    printf("%d\n", vetor[0]);
}

int main() {
    int v[3] = {200, 500, 300};

    muda_valor(v);

    printf("%d %d %d\n",
        v[0], v[1], v[2]);

    return 0;
}
```

```c
#include <stdio.h>

void muda_valor(int *vetor) {
    vetor[0] = 90;
    vetor[1] = 507;

    printf("%d\n", vetor[0]);
}

int main() {
    int v[3] = {200, 500, 300};

    muda_valor(v);

    printf("%d %d %d\n",
        v[0], v[1], v[2]);

    return 0;
}
```

Há diferença na saída dos dois programas?

```c
#include <stdio.h>

void muda_valor(int vetor[]) {
    vetor[0] = 90;
    vetor[1] = 507;

    printf("%d\n", vetor[0]);
}

int main() {
    int v[3] = {200, 500, 300};

    muda_valor(v);

    printf("%d %d %d\n",
        v[0], v[1], v[2]);

    return 0;
}
```

```c
#include <stdio.h>

void muda_valor(int *vetor) {
    vetor[0] = 90;
    vetor[1] = 507;

    printf("%d\n", vetor[0]);
}

int main() {
    int v[3] = {200, 500, 300};

    muda_valor(v);

    printf("%d %d %d\n",
        v[0], v[1], v[2]);

    return 0;
}
```

Há diferença na saída dos dois programas?

```
90
90 507 300
```

```
90
90 507 300
```

Retorno de vetor por função

- Para retornar um vetor, precisamos retornar seu ponteiro:

```
int* cria_vetor(int n) {

   // Implementacao da funcao

}
```

Retorno de vetor por função

Qual a saída deste programa?

```c
#include <stdio.h>

int* cria_vetor(int n) {
    int vetor[n];

    int i;
    for (i = 0; i < n; i++)
        vetor[i] = i+1;

    return vetor;
}


int main() {
    int *v = cria_vetor(5);

    int i;
    for (i = 0; i < 5; i++)
        printf("%d ", v[i]);
    printf("\n");

    return 0;
}
```

Retorno de vetor por função

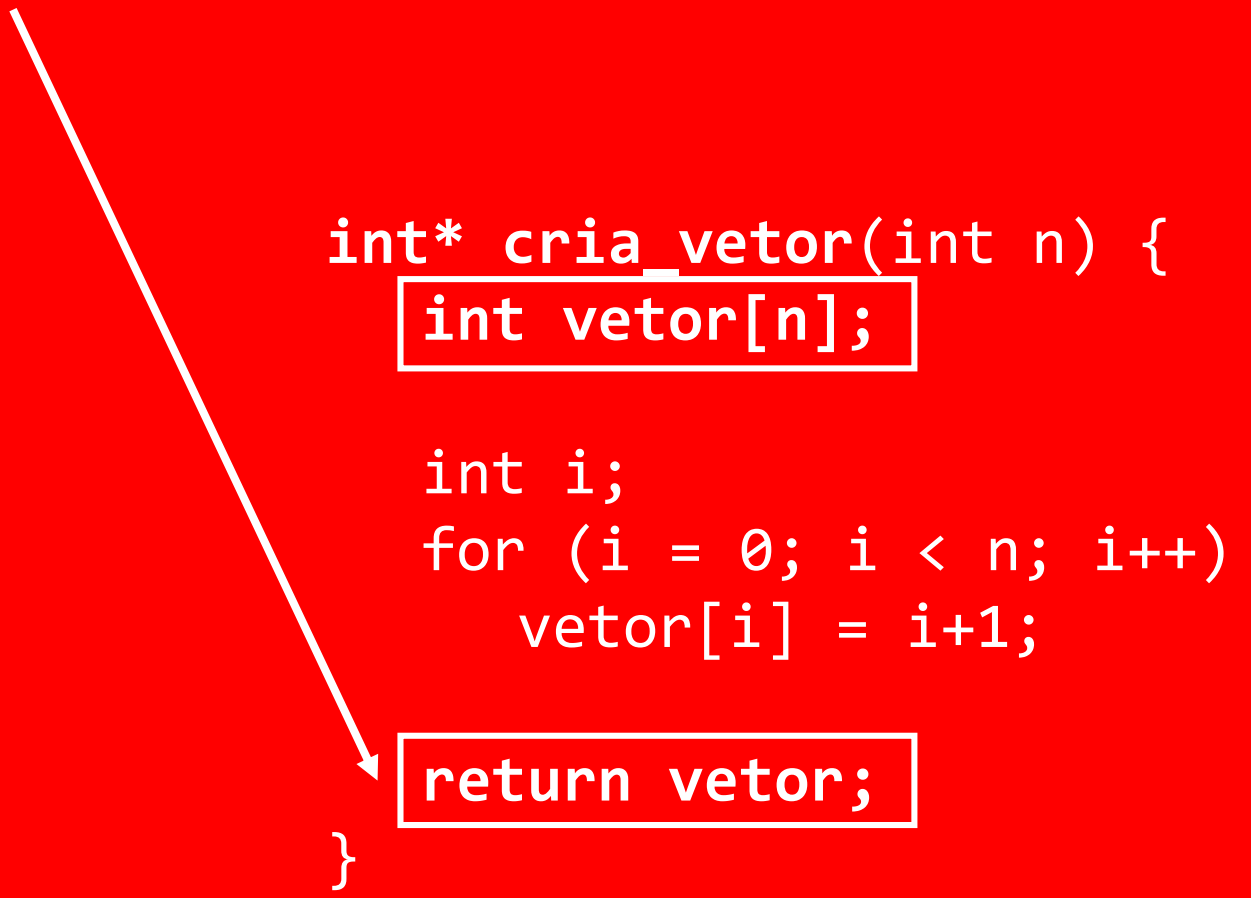
Função retornou ponteiro para variável local!

Qual a saída deste programa?

Segmentation fault (core dumped)

```c
#include <stdio.h>

int* cria_vetor(int n) {
    int vetor[n];

    int i;
    for (i = 0; i < n; i++)
        vetor[i] = i+1;

    return vetor;
}


int main() {
    int *v = cria_vetor(5);

    int i;
    for (i = 0; i < 5; i++)
        printf("%d ", v[i]);
    printf("\n");

    return 0;
}
```

Não retorne ponteiro para variável local!



```
int* cria_vetor(int n) {
    int vetor[n];

    int i;
    for (i = 0; i < n; i++)
        vetor[i] = i+1;

    return vetor;
}
```

Retorno de vetor por função

Qual a saída deste programa?

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

int* cria_vetor(int n) {
    int *vetor = malloc(sizeof(int) * n);

    int i;
    for (i = 0; i < n; i++)
        vetor[i] = i+1;

    return vetor;
}

int main() {
    int *v = cria_vetor(5);

    int i;
    for (i = 0; i < 5; i++)
        printf("%d ", v[i]);
    printf("\n");
    free(v);

    return 0;
}
```

Retorno de vetor por função

Qual a saída deste programa?

1 2 3 4 5

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

int* cria_vetor(int n) {
    int *vetor = malloc(sizeof(int) * n);

    int i;
    for (i = 0; i < n; i++)
        vetor[i] = i+1;

    return vetor;
}

int main() {
    int *v = cria_vetor(5);

    int i;
    for (i = 0; i < 5; i++)
        printf("%d ", v[i]);
    printf("\n");
    free(v);

    return 0;
}
```

Mais ponteiros

- Mais detalhes, exemplos e exercícios podem ser encontrados no material extra disponível no site da disciplina;

- **É recomendável estudar esse material extra!**

Referências

- Slides do Prof. Paulo H. Pisani: Programação Estruturada (Q3/2018)
- Slides do Prof. Jesús P. Mena-Chalco:
  - http://professor.ufabc.edu.br/~jesus.mena/courses/mcta028-3q-2017/
- Slides do Prof. Fabrício Olivetti:
  - http://folivetti.github.io/courses/ProgramacaoEstruturada/
- PINHEIRO, F. A. C. Elementos de programação em C. Porto Alegre, RS: Bookman, 2012.
- CELES, W.; CERQUEIRA, R.; RANGEL, J. L. Introdução a Estruturas de Dados. Elsevier/Campus, 2004.