

Ponteiros

Prof. Paulo Henrique Pisani

fevereiro/2022

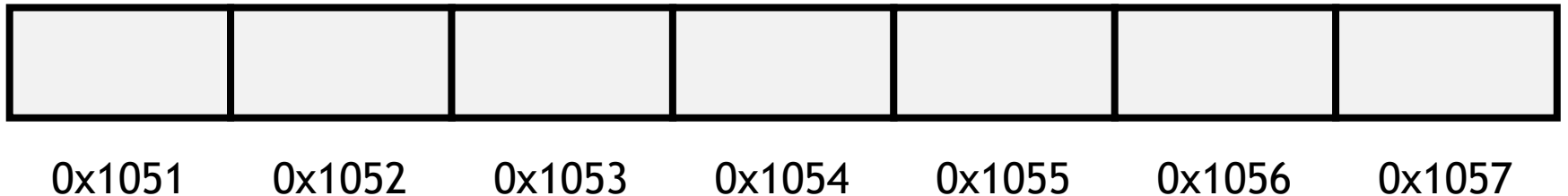
Tópicos

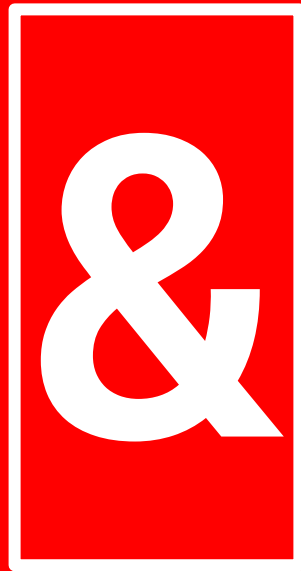
- Memória, endereços e ponteiros;
- Passagem de parâmetros por referência;
- Alocação estática vs Alocação dinâmica;
- Aritmética de ponteiros;
- Vetores como parâmetro e como retorno de função;
- Ponteiro para ponteiro.

Memória, Endereços e Ponteiros

Memória

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





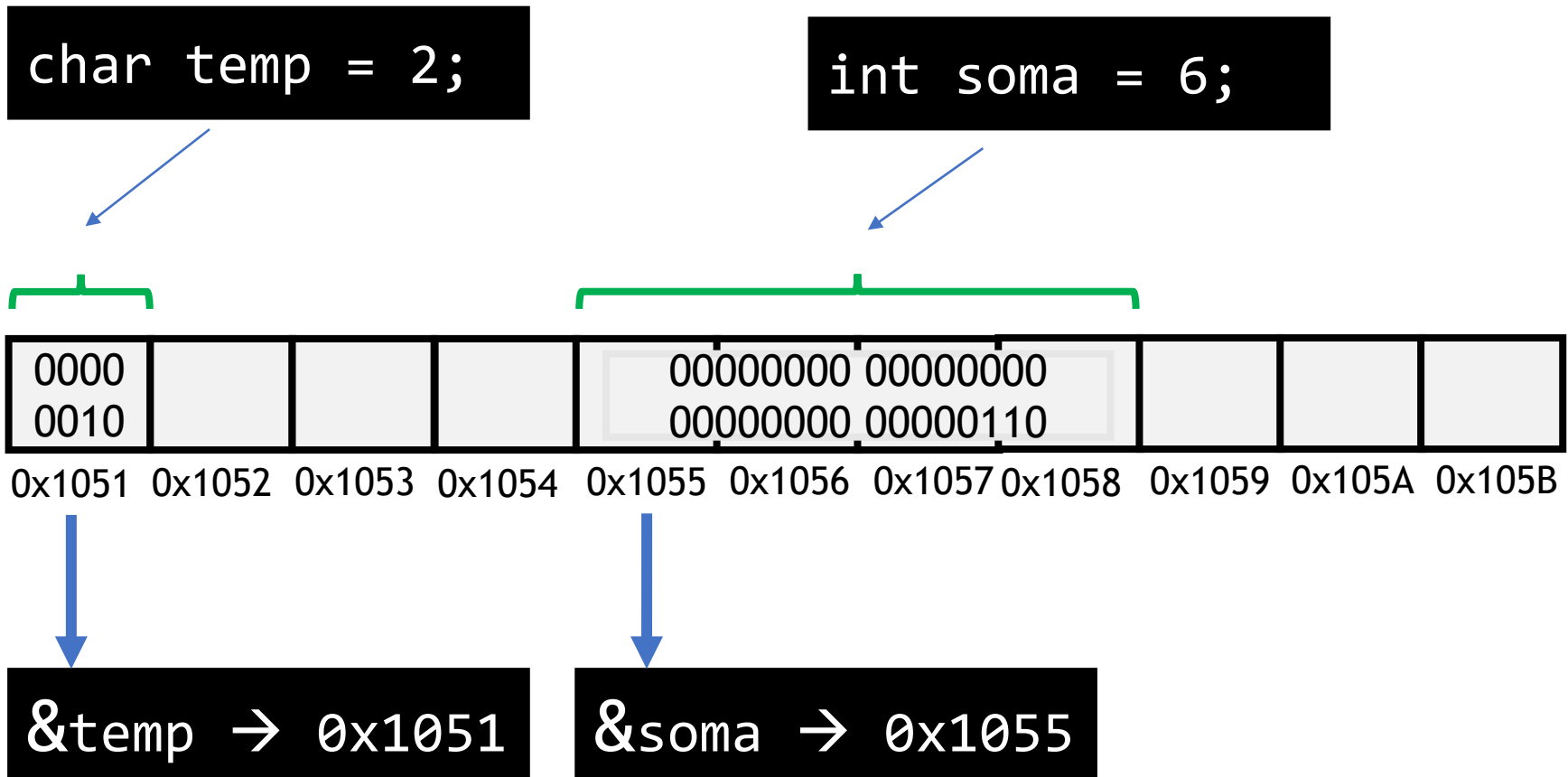
Este é o operador address-of!
Ele retorna o endereço do item a
sua direita!

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

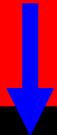
- Ponteiro é uma variável que armazena um endereço de memória;
- Para declarar um ponteiro basta incluir um **asterisco** antes do nome da variável:



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


Ponteiro

- 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;
```

```
#include <stdio.h>

int main() {
    char var1;
    int var2;
    double var3;

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

    printf("Tamanhos %ld %ld %ld\n",
           sizeof(var1),
           sizeof(var2),
           sizeof(var3));

    printf("Tamanhos %ld %ld %ld\n",
           sizeof(ponteiro1),
           sizeof(ponteiro2),
           sizeof(ponteiro3));

    return 0;
}
```

O que será
impresso?

```
#include <stdio.h>

int main() {
    char var1;
    int var2;
    double var3;

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

    printf("Tamanhos %ld %ld %ld\n",
           sizeof(var1),
           sizeof(var2),
           sizeof(var3));

    printf("Tamanhos %ld %ld %ld\n",
           sizeof(ponteiro1),
           sizeof(ponteiro2),
           sizeof(ponteiro3));

    return 0;
}
```

O que será
impresso?

**Depende: 32-bit
ou 64-bit**

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

```
int main() {  
    char var1;  
    int var2;  
    double var3;  
  
    char *ponteiro1;  
    int *ponteiro2;  
    double *ponteiro3;  
  
    printf("Tamanhos %ld %ld %ld\n",  
          sizeof(var1),  
          sizeof(var2),  
          sizeof(var3));  
  
    printf("Tamanhos %ld %ld %ld\n",  
          sizeof(ponteiro1),  
          sizeof(ponteiro2),  
          sizeof(ponteiro3));  
  
    return 0;  
}
```

O que será
impresso?

32-bit:

Tamanhos	1	4	8
Tamanhos	4	4	4

64-bit:

Tamanhos	1	4	8
Tamanhos	8	8	8

Ponteiro

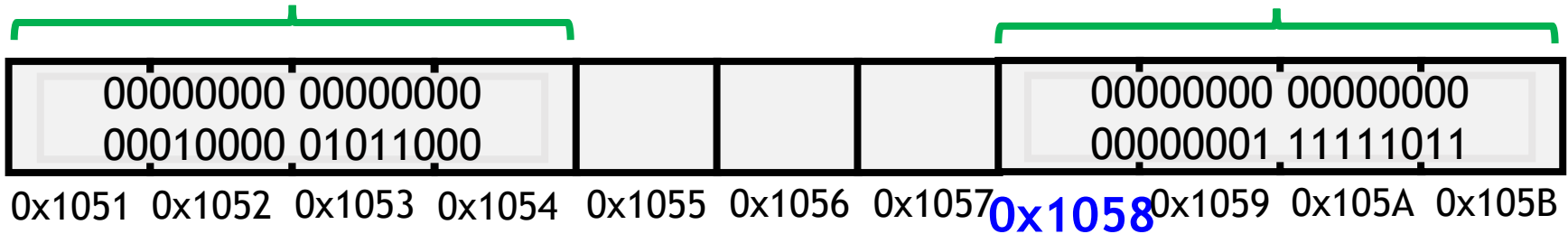
```
int n = 507;  
int *ptr;  
ptr = &n;
```

Ponteiro 32-bit

ptr = 0x1058



n = 507



Endereço do ponteiro

- Um ponteiro é uma variável, portanto, ele armazena um endereço de memória.

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

```
int main() {
```

```
    int num = 507;
```

```
    int *pt1 = &num;
```

```
    printf("%d %p %p %p\n", num, &num, pt1, &pt1);
```

```
    return 0;
```

```
}
```

O que será impresso?

Endereço do ponteiro

- Um ponteiro é uma variável, portanto, ele armazena um endereço de memória.

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

```
int main() {
```

```
    int num = 507;
```

```
    int *pt1 = &num;
```

```
    printf("%d %p %p %p\n", num, &num, pt1, &pt1);
```

```
    return 0;
```

```
}
```

O que será impresso?

507 0x7fff9578caac 0x7fff9578caac 0x7fff9578cab0

É possível acessar o valor da variável
a partir do endereço de memória?

É possível acessar o valor da variável a partir do endereço de memória?

Sim, para isso usamos o próprio
asterisco: *

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*

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 = &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

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

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?

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?

Teste 3

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

O que será
impresso?

Teste 4

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

O que será
impresso?

Passagem de
parâmetros por
referência

Lembram desse slide?



Passagem de parâmetros

- Em C, todo parâmetro de função é passado **por valor**;
- Para passar um argumento **por referência**, precisamos passar o valor do endereço de memória (ponteiro) - veremos isso em outras aulas...

Parâmetros e argumentos são a mesma coisa?

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”

```
#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?

```
#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
```



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

```
void troca_valor_a(double a, double b) {  
    double temp = a;  
    a = b;  
    b = temp;  
}
```

```
void troca_valor_b(double *a, double *b) {  
    double temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

```
int main() {  
    double n1=10, n2 = 20;  
    printf("%.21f %.21f\n", n1, n2);  
  
    troca_valor_a(n1, n2);  
    printf("%.21f %.21f\n", n1, n2);  
  
    troca_valor_b(&n1, &n2);  
    printf("%.21f %.21f\n", n1, n2);  
    return 0;  
}
```

Qual a saída
desse programa?

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

```
void troca_valor_a(double a, double b) {  
    double temp = a;  
    a = b;  
    b = temp;  
}
```

```
void troca_valor_b(double *a, double *b) {  
    double temp = *a;  
    *a = *b;  
    *b = temp;  
}
```


```
int main() {  
    double n1=10, n2 = 20;  
    printf("%.21f %.21f\n", n1, n2);  
  
    troca_valor_a(n1, n2);  
    printf("%.21f %.21f\n", n1, n2);  
  
    troca_valor_b(&n1, &n2);  
    printf("%.21f %.21f\n", n1, n2);  
    return 0;  
}
```

Qual a saída
desse programa?

```
10.00 20.00  
10.00 20.00  
20.00 10.00
```

Retornando mais de um valor

- Para isso, passamos parâmetros por referência



```
void divide(int dividendo, int divisor, int *quociente, int *resto) {  
    *quociente = dividendo / divisor;  
    *resto = dividendo % divisor;  
}
```

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

```
void divide(int dividendo, int divisor, int *quociente, int *resto) {  
    *quociente = dividendo / divisor;  
    *resto = dividendo % divisor;  
}
```

```
int main() {  
  
    int a, b;  
    scanf("%d %d", &a, &b);  
  
    int q, r;  
    divide(a, b, &q, &r);  
  
    printf("q=%d r=%d", q, r);  
  
    return 0;  
}
```

Alocação estática vs Alocação dinâmica

Alocação dinâmica

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

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

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

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

Alocação dinâmica

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

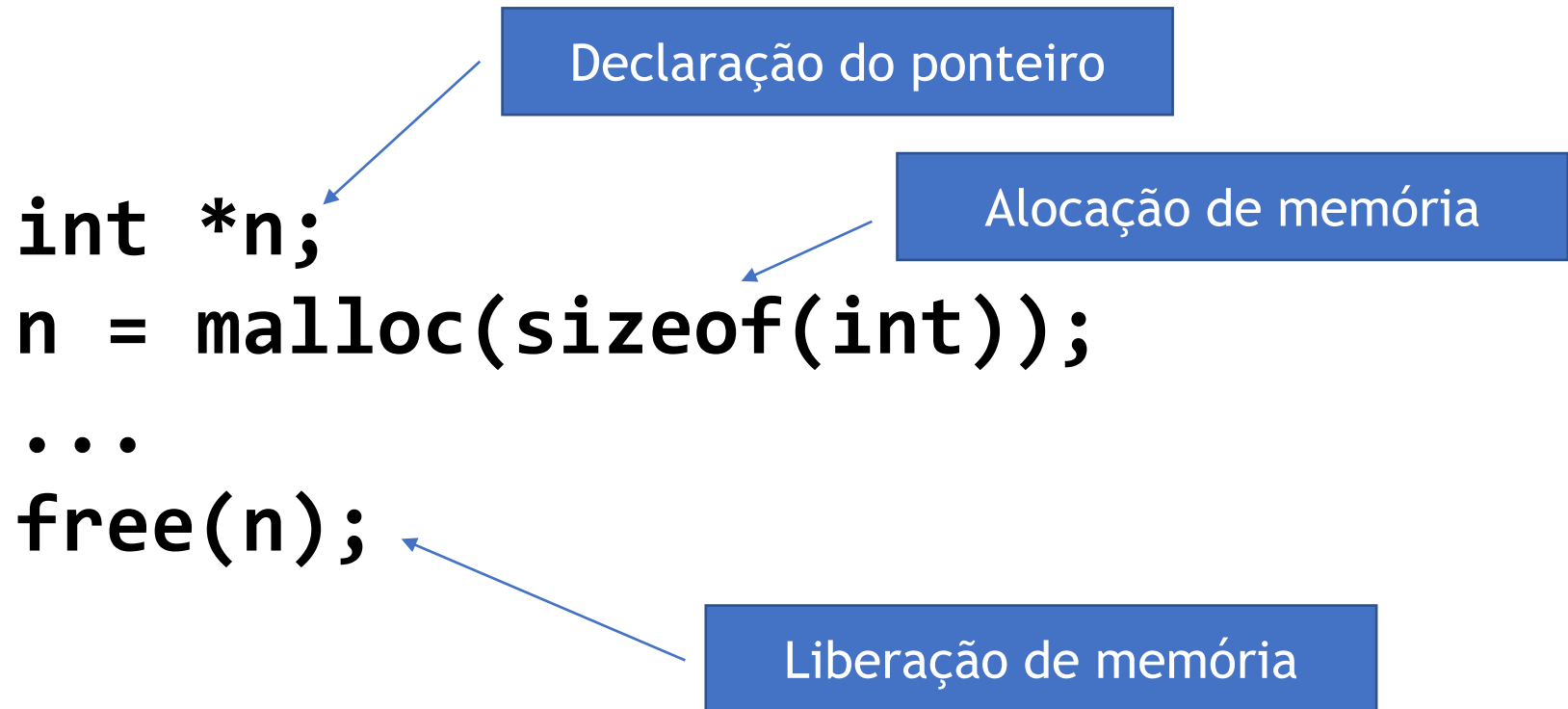
```
void* malloc( size_t size );
```

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

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

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

Alocação dinâmica



Alocação dinâmica

```
int *n;
```

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

```
...
```

```
free(n);
```

Podemos usar um cast também



Lembre-se de sempre liberar a memória alocada!

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

`free(n);`

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

```
int main() {
    int *n;
    n = malloc(sizeof(int));

    if (n != NULL) {
        *n = 507;
        printf("%d\n", *n);
        free(n);
    } else {
        printf("Erro na alocação.\n");
    }

    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)

Valgrind

- Ferramenta de análise que pode detectar vazamentos de memória (*memory leaks*), acessos a áreas de memória indevidas, etc.

<http://valgrind.org>

Valgrind

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

int main() {
    int *n;
    n = malloc(sizeof(int));

    if (n != NULL) {
        *n = 507;
        printf("%d\n", *n);
        free(n);
    } else {
        printf("Erro na alocação.\n");
    }

    return 0;
}
```



```
$ valgrind ./teste.exe
==53== Memcheck, a memory error detector
==53== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==53== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright
info
==53== Command: ./t1.exe
==53==
==53== error calling PR_SET_PTRACER, vgdb might block
507
==53==
==53== HEAP SUMMARY:
==53==   in use at exit: 0 bytes in 0 blocks
==53==   total heap usage: 2 allocs, 2 frees, 516 bytes allocated
==53==
==53== All heap blocks were freed -- no leaks are possible
==53==
==53== For counts of detected and suppressed errors, rerun with: -v
==53== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Removemos o free. E agora?

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

int main() {
    int *n;
    n = malloc(sizeof(int));

    if (n != NULL) {
        *n = 507;
        printf("%d\n", *n);
        /*free(n);*/
    } else {
        printf("Erro na alocação.\n");
    }

    return 0;
}
```



```
$ valgrind ./teste.exe
```

```
==59== Memcheck, a memory error detector
==59== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==59== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright
info
==59== Command: ./t1.exe
==59==
==59== error calling PR_SET_PTRACER, vgdb might block
507
==59==
==59== HEAP SUMMARY:
==59==    in use at exit: 4 bytes in 1 blocks
==59==    total heap usage: 2 allocs, 1 frees, 516 bytes allocated
==59==
==59== LEAK SUMMARY:
==59==    definitely lost: 4 bytes in 1 blocks
==59==    indirectly lost: 0 bytes in 0 blocks
==59==    possibly lost: 0 bytes in 0 blocks
==59==    still reachable: 0 bytes in 0 blocks
==59==    suppressed: 0 bytes in 0 blocks
==59== Rerun with --leak-check=full to see details of leaked memory
==59==
==59== For counts of detected and suppressed errors, rerun with: -v
==59== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

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

int conta_a(char texto[], int max) {
    int *i = (int *) malloc(sizeof(int));
    int c;
    for (*i = 0; *i < max; (*i)++) {
        if (texto[*i] == 0)
            return c;
        if ((texto[*i] == 'A') || (texto[*i] == 'a'))
            c++;
    }
    free(i);
    return c;
}

int main() {
    char texto[50] = "UFABC_UFABC_abc";

    int c = conta_a(texto, 50);
    printf("%d\n", c);

    return 0;
}
```

Este programa funciona? E se funciona, pode ocorrer vazamento de memória?

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

```
int conta_a(char texto[], int max) {
    int *i = (int *) malloc(sizeof(int));
    int c;
    for (*i = 0; *i < max; (*i)++) {
        if (texto[*i] == 0)
            return c;
        if ((texto[*i] == 'A') || (texto[*i] == 'a'))
            c++;
    }
    free(i);
    return c;
}
```

```
int main() {
    char texto[50] = "UFABC_UFABC_abc";

    int c = conta_a(texto, 50);
    printf("%d\n", c);

    return 0;
}
```

Erro: i é um ponteiro, mas o que queremos é o valor do inteiro no endereço i. Para corrigir, basta adicionar o *

Este programa funciona? E se funciona, pode ocorrer vazamento de memória?

malloc e calloc

- Além da função malloc:

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

- Há também a função calloc:

```
void* calloc(size_t nitems, size_t size);
```

- **malloc** apenas aloca um bloco de memória (mas não inicializa a memória);
- **calloc** aloca e inicializa o bloco com zeros.

scanf

- scanf recebe um ponteiro, certo?


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

int main() {
    int *ptr_numero;

    scanf("%d", ptr_numero);

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

    return 0;
}
```



Há algo errado
neste programa?

scanf

- scanf recebe um ponteiro, certo?

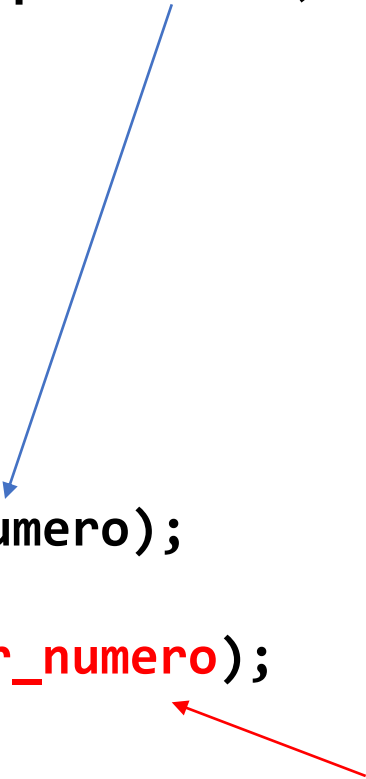
```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *ptr_numero;

    scanf("%d", ptr_numero);

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

    return 0;
}
```



Faltou acessar o
valor do inteiro
apontado

scanf

- scanf recebe um ponteiro, certo?

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

```
int main() {
    int *ptr_numero;

    scanf("%d", ptr_numero);

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

    return 0;
}
```

Além disso, o valor do ponteiro ptr_numero está indefinido!

scanf

- `scanf` recebe um ponteiro, certo?

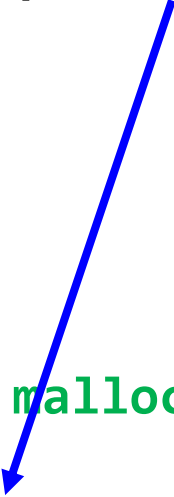
```
#include <stdio.h>
#include <stdlib.h>
```

```
int main() {
    int *ptr_numero = malloc(sizeof(int));

    scanf("%d", ptr_numero);

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

    return 0;
}
```



scanf

- scanf recebe um ponteiro, certo?

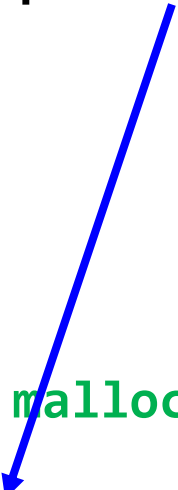
```
#include <stdio.h>
#include <stdlib.h>
```

```
int main() {
    int *ptr_numero = malloc(sizeof(int));

    scanf("%d", ptr_numero);

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

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



Alocação estática vs Alocação dinâmica

(vetores)

Alocação dinâmica

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

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

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

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

Alocação dinâmica

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

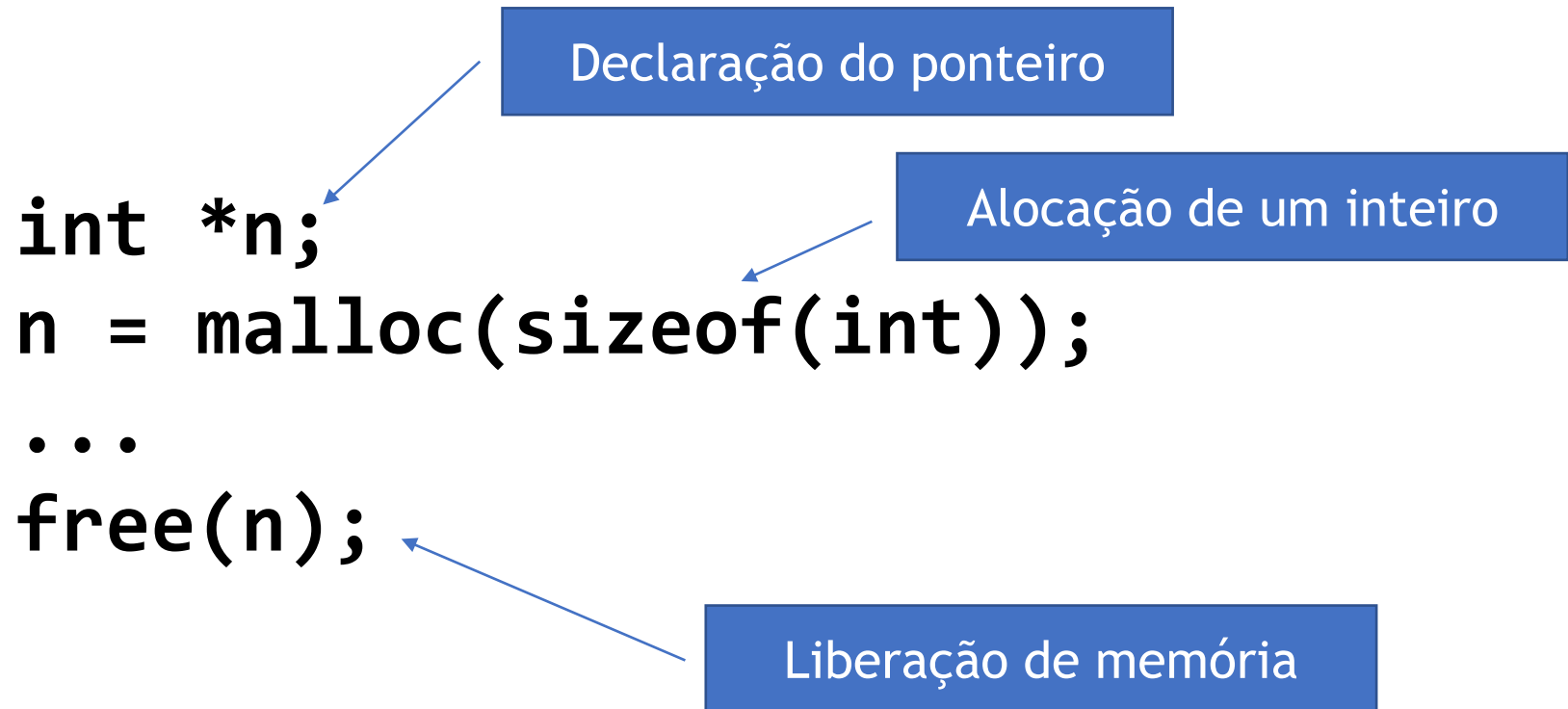
```
void* malloc( size_t size );
```

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

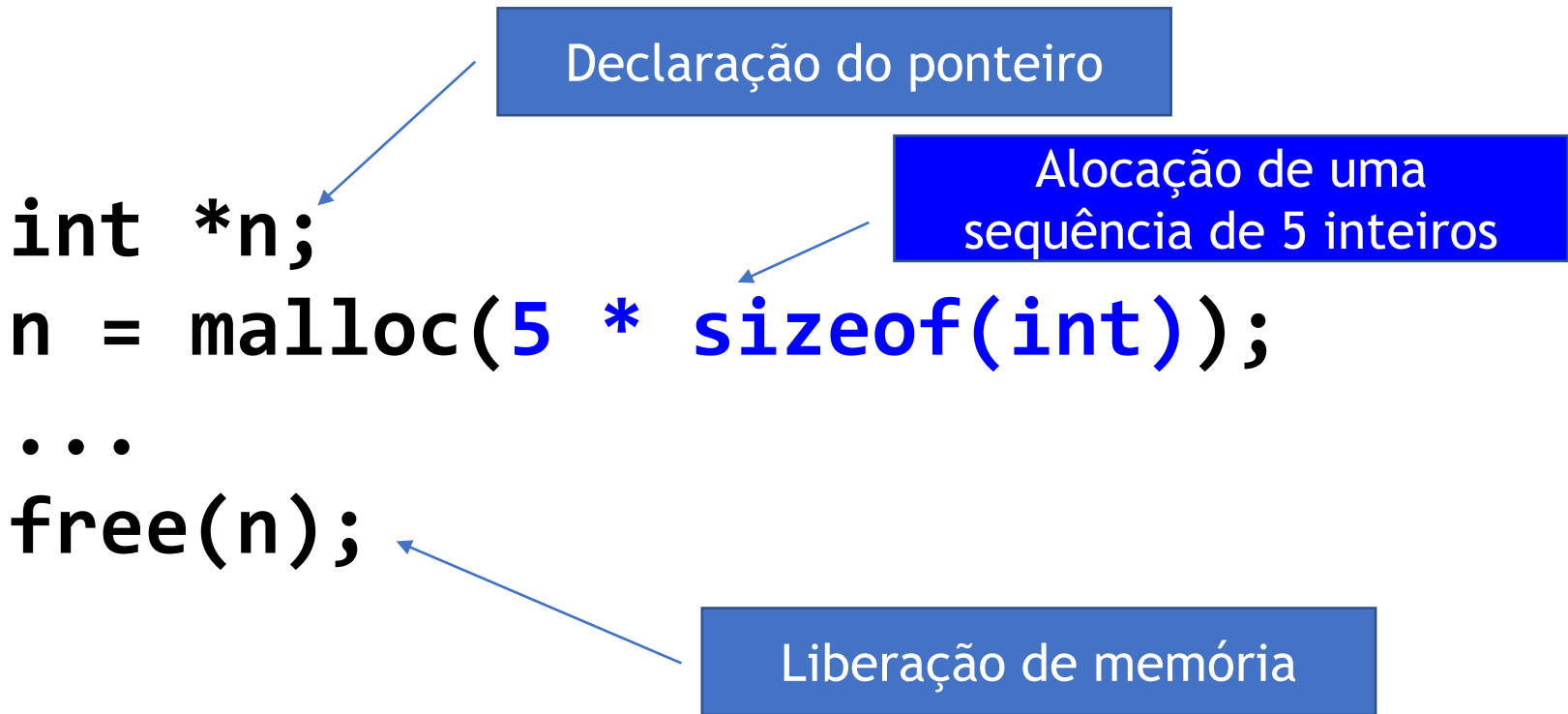
```
void free( void* ptr );
```

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

Alocação dinâmica



Alocação dinâmica (vetores)



Alocação dinâmica (vetores)

```
#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;
}
```

```
#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;
}
```

Lembre-se de sempre liberar a memória alocada!

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


`free(n);`

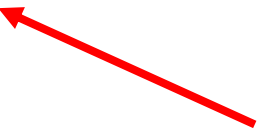
```
#include<stdio.h>
#include<stdlib.h>
```

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

    int *vetor = malloc(sizeof(int) * n);
    if (vetor == NULL) {
        printf("Erro na alocao.\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)

Aritmética de ponteiros

Aritmética de ponteiros

- Podemos utilizar alguns operadores aritméticos sobre ponteiros:

a) ++

b) --

c) +

d) -

Aritmética de ponteiros

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

```
int main() {
    char *ptr = malloc(sizeof(char));

    printf("%p\n", ptr);

    ptr++;

    printf("%p\n", ptr);

    return 0;
}
```

O que será impresso
no segundo printf?

Saída

00740D00

?

Aritmética de ponteiros

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

```
int main() {
    char *ptr = malloc(sizeof(char));

    printf("%p\n", ptr);

    ptr++;

    printf("%p\n", ptr);

    return 0;
}
```

O que será impresso
no segundo printf?

Saída

00740D00
00740D01

Incrementou apenas uma unidade!

Aritmética de ponteiros

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

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

    printf("%p\n", ptr);

    ptr++;

    printf("%p\n", ptr);

    return 0;
}
```

O que será impresso
no segundo printf?

Saída

00750D00

?

Aritmética de ponteiros

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

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

    printf("%p\n", ptr);

    ptr++;

    printf("%p\n", ptr);

    return 0;
}
```

O que será impresso
no segundo printf?

Saída

00750D00
00750D04

Incrementou 4 unidades!



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       $\longleftrightarrow$       vetor[0] = 80
```

```
*(vetor + 4) = 507       $\longleftrightarrow$       vetor[4] = 507
```

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

Vetores como
parâmetro e como
retorno de função

Passagem de vetor como parâmetro

- Já vimos que 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;
}
```

```
#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;
}
```

```
#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?


```
#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;
}
```

```
#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
```

```
#include <stdio.h>

void muda_valor(int vetor[]) {
    *vetor = 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;
}
```

```
#include <stdio.h>

void muda_valor(int *vetor) {
    *vetor = 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?

```

#include <stdio.h>

void muda_valor(int vetor[]) {
    *vetor = 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;
}

```

```

#include <stdio.h>

void muda_valor(int *vetor) {
    *vetor = 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

- Qual a melhor forma de retornar um vetor?

```
int[] cria_vetor(int n) {  
    int vetor[n];  
  
    int i;  
    for (i = 0; i < n; i++)  
        vetor[i] = i+1;  
  
    return vetor;  
}
```

A

```
int[n] cria_vetor(int n) {  
    int vetor[n];  
  
    int i;  
    for (i = 0; i < n; i++)  
        vetor[i] = i+1;  
  
    return vetor;  
}
```

B

Retorno de vetor por função

- Qual a melhor forma de retornar um vetor?


```
int[] cria_vetor(int n) {  
    int vetor[n];  
  
    int i;  
    for (i = 0; i < n; i++)  
        vetor[i] = i+1;  
  
    return vetor;  
}
```

```
int* cria_vetor(int n) {  
    int vetor[n];  
  
    int i;  
    for (i = 0; i < n; i++)  
        vetor[i] = i+1;  
  
    return vetor;  
}
```

Nenhuma das duas formas está correta!

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?

```
#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)

```
#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?

```
#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

```
#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;
}
```

Ponteiro para
ponteiro

Ponteiro para ponteiro

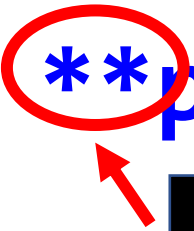
- Ponteiro é uma variável que armazena endereços de memória;
- Um ponteiro para ponteiro armazena o endereço de memória de um ponteiro.

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

Ponteiro para ponteiro

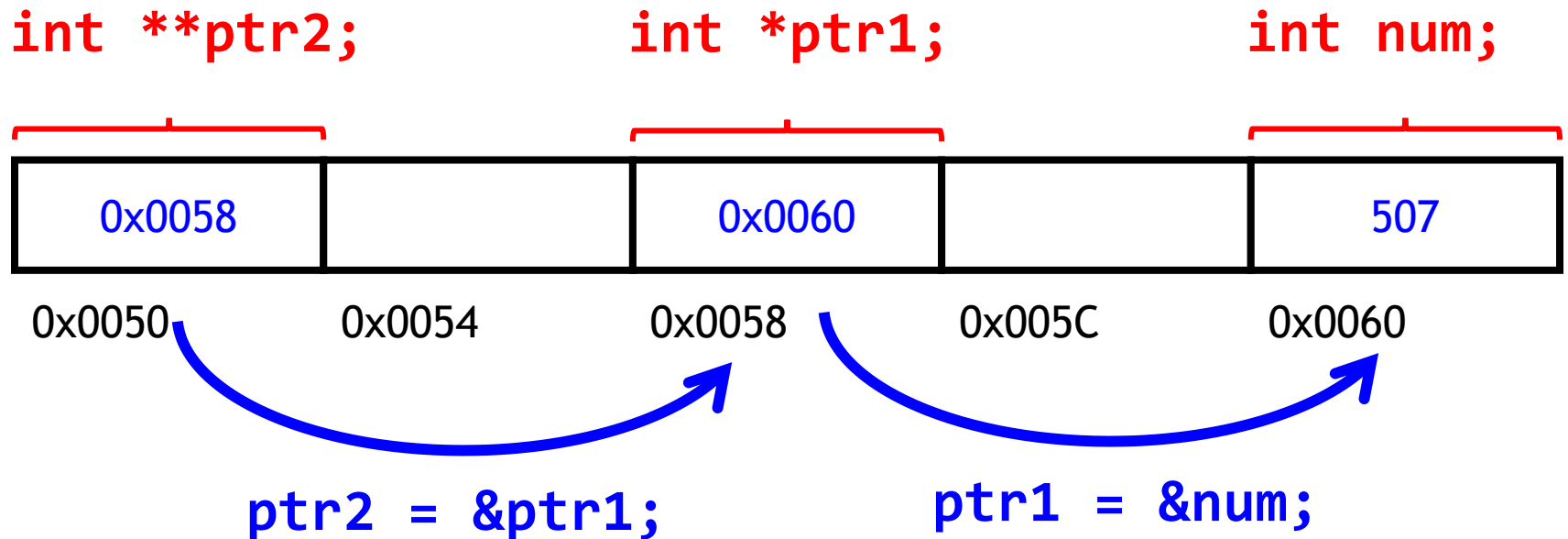
- Ponteiro é uma variável que armazena endereços de memória;
- Um ponteiro para ponteiro armazena o endereço de memória de um ponteiro.

```
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() {
```

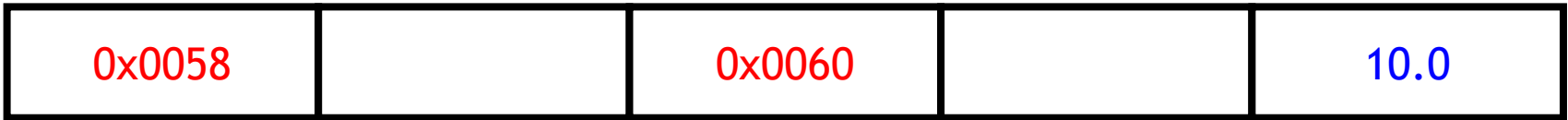
```
    double nota_p1;  
    scanf("%lf", &nota_p1);  
    printf("%.2lf\n", nota_p1);
```

```
    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>

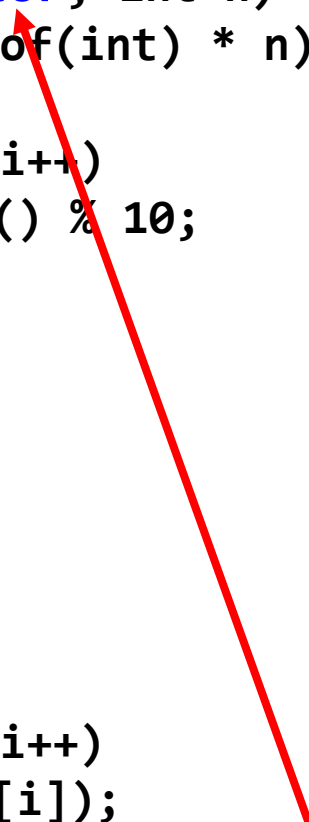
void le_vetor(int *vetor, int n) {
    vetor = malloc(sizeof(int) * n);
    int i;
    for (i = 0; i < n; i++)
        vetor[i] = rand() % 10;
}

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;
}
```

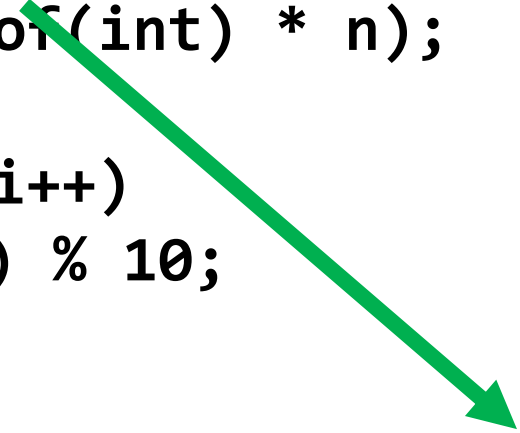


O que será
impresso?

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

Referências

- 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/>

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.
- KNUTH, D. E. The art of computer programming. Upper Saddle River, USA: Addison-Wesley, 2005.
- SEDGEWICK, R. Algorithms in C: parts 1-4 (fundamental algorithms, data structures, sorting, searching). Reading, USA: Addison-Wesley, 1998.

Bibliografia complementar

- DROZDEK, A. Estrutura de dados e algoritmos em C++. São Paulo, SP: Thomson Learning, 2002.
- RODRIGUES, P.; PEREIRA, P.; SOUSA, M. Programação em C++: conceitos básicos e algoritmos. Lisboa, PRT: FCA de Informática, 2000.
- SZWARCFITER, J. L.; MARKENZON, L. Estruturas de dados e seus algoritmos. 3. ed. Rio de Janeiro, RJ: LTC, 2010.
- TENENBAUM, A. M.; LANGSAM Y.; AUGENSTEIN M. J. Estruturas de dados usando C. São Paulo, SP: Pearson Makron Books, 1995.
- ZIVIANI, N. Projeto de algoritmos com implementação em Java e C++. São Paulo, SP: Thomson Learning, 2007.