

# Ponteiros (parte 1)

Prof. Paulo Henrique Pisani

outubro/2021

# Tópicos

- Memória, endereços e ponteiros
- Passagem de parâmetros por referência
- Alocação estática vs Alocação dinâmica

Memória, endereços e  
ponteiros

# Memória

- Estrutura de um programa carregado em memória:



- **Instruções (código):** código binário do programa;
- **Dados estáticos:** variáveis globais e estáticas (existem durante toda a execução do programa).

# Memória

- Estrutura de um programa carregado em memória:



- **Heap:** variáveis criadas por alocação dinâmica;
- **Pilha:** variáveis locais criadas para a execução de uma função (são removidas após o término da função).

# size

- Comando para mostrar o tamanho das seções de um arquivo binário (incluído no pacote no MinGW também);
- Veremos alguns exemplos a seguir...

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

```
int main() {  
    return 0;  
}
```

text	data	bss	dec	hex	filename
1415	544	8	1967	7af	t1.exe

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

```
double g_variavel;
```

```
int main() {  
    return 0;  
}
```

Dados não inicializados que receberão o valor zero

text	data	bss	dec	hex	filename
1415	544	16	1975	7b7	t1.exe

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

```
double g_variavel = 507.0;
```

```
int main() {  
    return 0;  
}
```

Agora a variável está inicializada

text	data	bss	dec	hex	filename
1415	552	8	1975	7b7	t1.exe

### Programa A

```
#include <stdio.h>

int a, b;

void funcao(void) {
    printf("%d %d\n", a, b);
}

int main(void) {
    funcao();
}
```

### Programa B

```
#include <stdio.h>

void funcao(void) {
    int a, b;

    printf("%d %d\n", a, b);
}

int main(void) {
    funcao();
}
```

Qual a saída desses programas?



## Programa A

```
#include <stdio.h>

int a, b;

void funcao(void) {
    printf("%d %d\n", a, b);
}

int main(void) {
    funcao();
}
```



0 0

Variável global (em bss) é  
inicializada com zero!

## Programa B

```
#include <stdio.h>

void funcao(void) {
    int a, b;

    printf("%d %d\n", a, b);
}

int main(void) {
    funcao();
}
```



-1490193088 22004

Comportamento imprevisível!  
Variáveis locais não são  
inicializadas!

# bss

- Dados no **bss** são automaticamente inicializados com zero!
- Variáveis globais estão no bss.

## Programa A

```
#include <stdio.h>

int a, b;

void funcao(void) {
    printf("%d %d\n", a, b);
}

int main(void) {
    funcao();
}
```



0 0

text	data	bss	dec	hex
1567	600	16	2183	887

## Programa B

```
#include <stdio.h>

void funcao(void) {
    int a, b;

    printf("%d %d\n", a, b);
}

int main(void) {
    funcao();
}
```

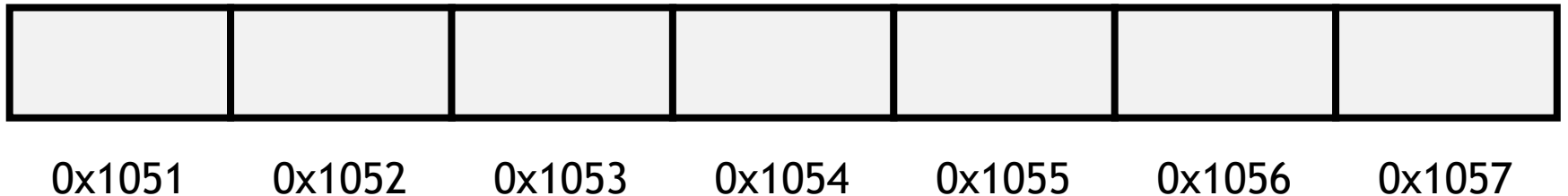


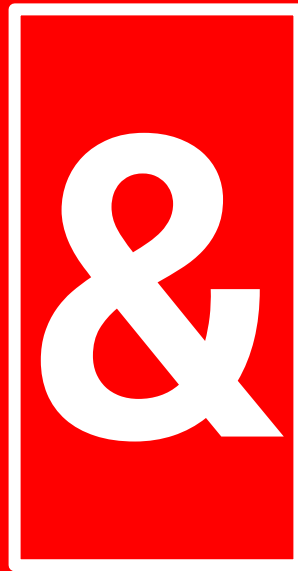
-1490193088 22004

text	data	bss	dec	hex
1567	600	8	2175	87f

# Memória

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





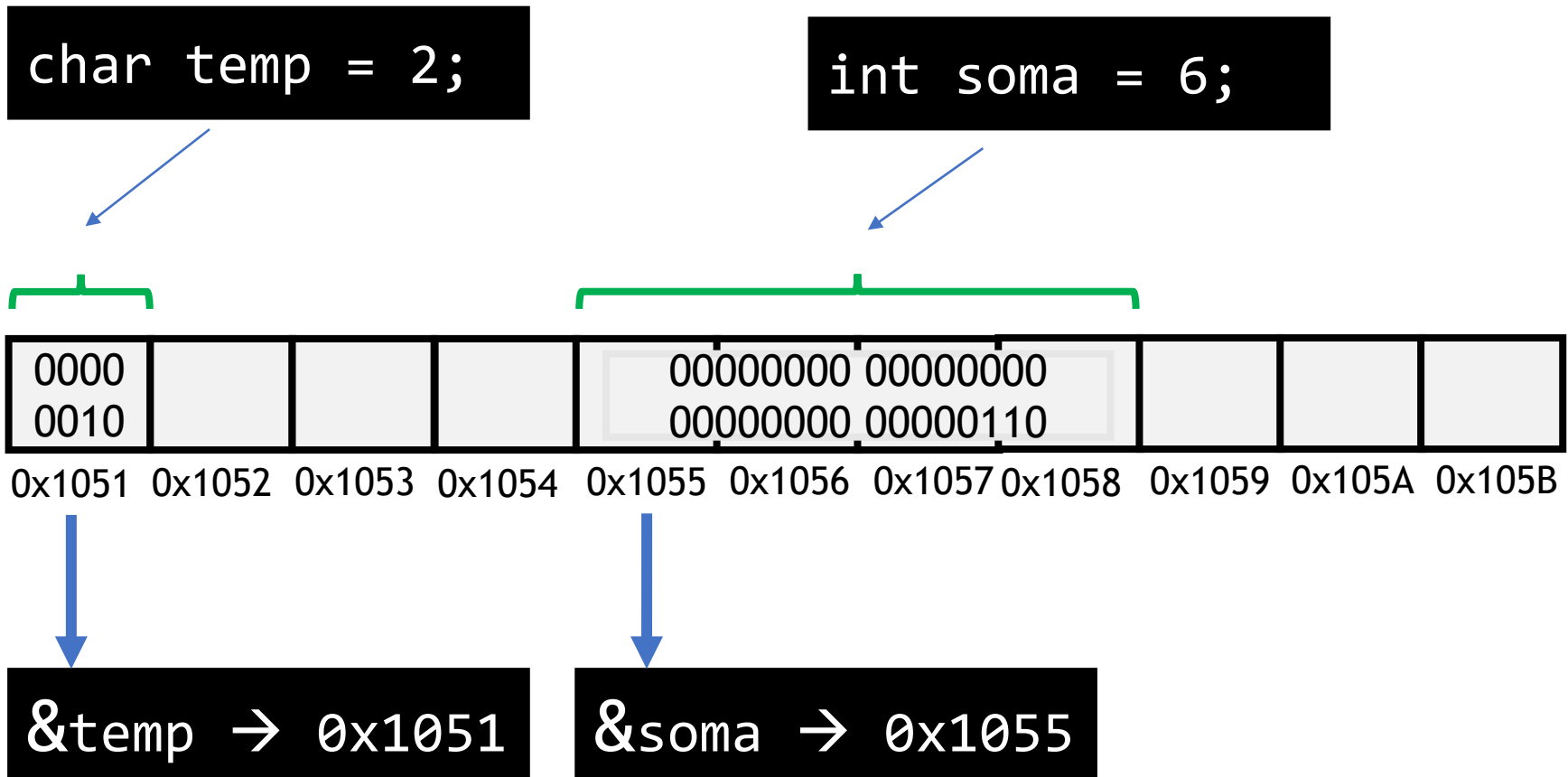
Este é o operador address-of!  
Ele retorna o endereço do item a  
sua direita!

Por exemplo:

**&temp** retorna o endereço de temp

**&soma** retorna o endereço de soma

# Endereço de uma variável



# Endereço de uma variável

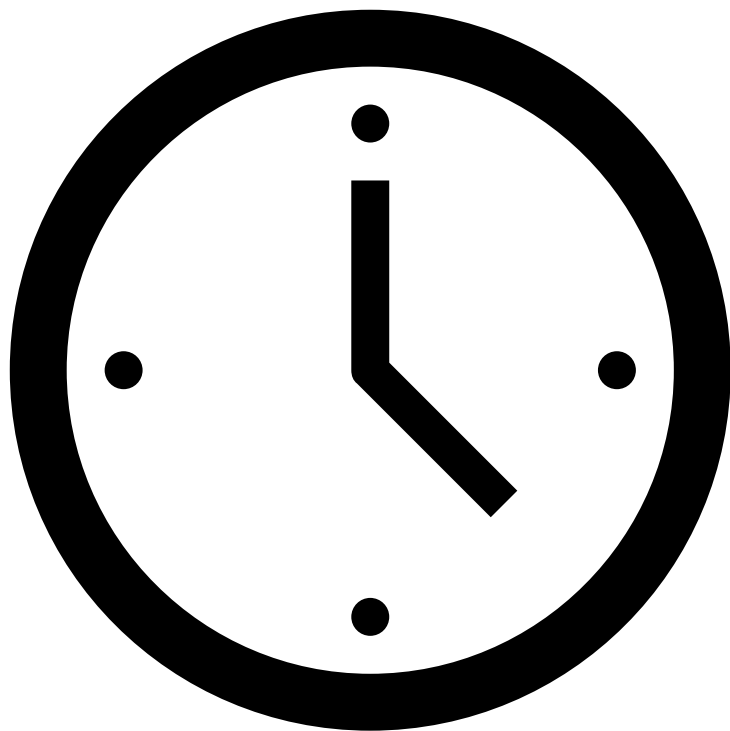
```
#include <stdio.h>
```

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



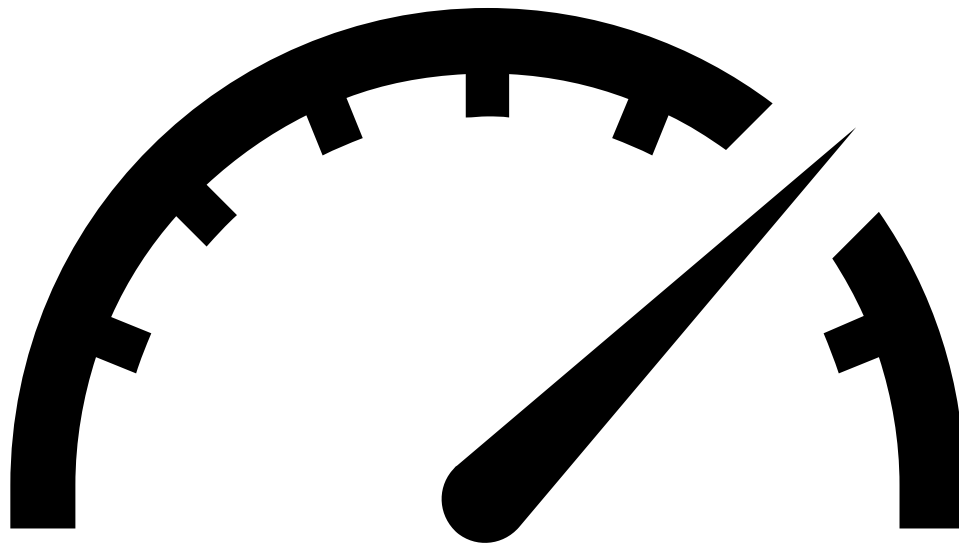
**Endereco do num=800 eh 0060FF0C**

# Ponteiro





# Ponteiro



# Ponteiro

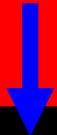
- Ponteiro é uma variável que armazena um endereço de memória;
- Para declarar um ponteiro basta incluir um **asterisco** antes do nome da variável:



```
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

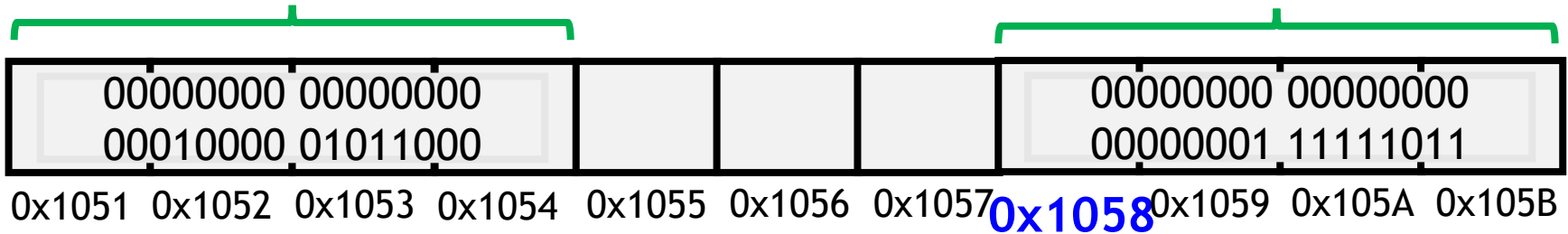
# Ponteiro

```
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?

# Teste 5

```
int v[10];  
int a = 507;  
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 v[10];  
int a = 507;  
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 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
```

# As duas versões são equivalentes?

Versão A

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

Versão B

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

# 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;  
}
```

# 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**:

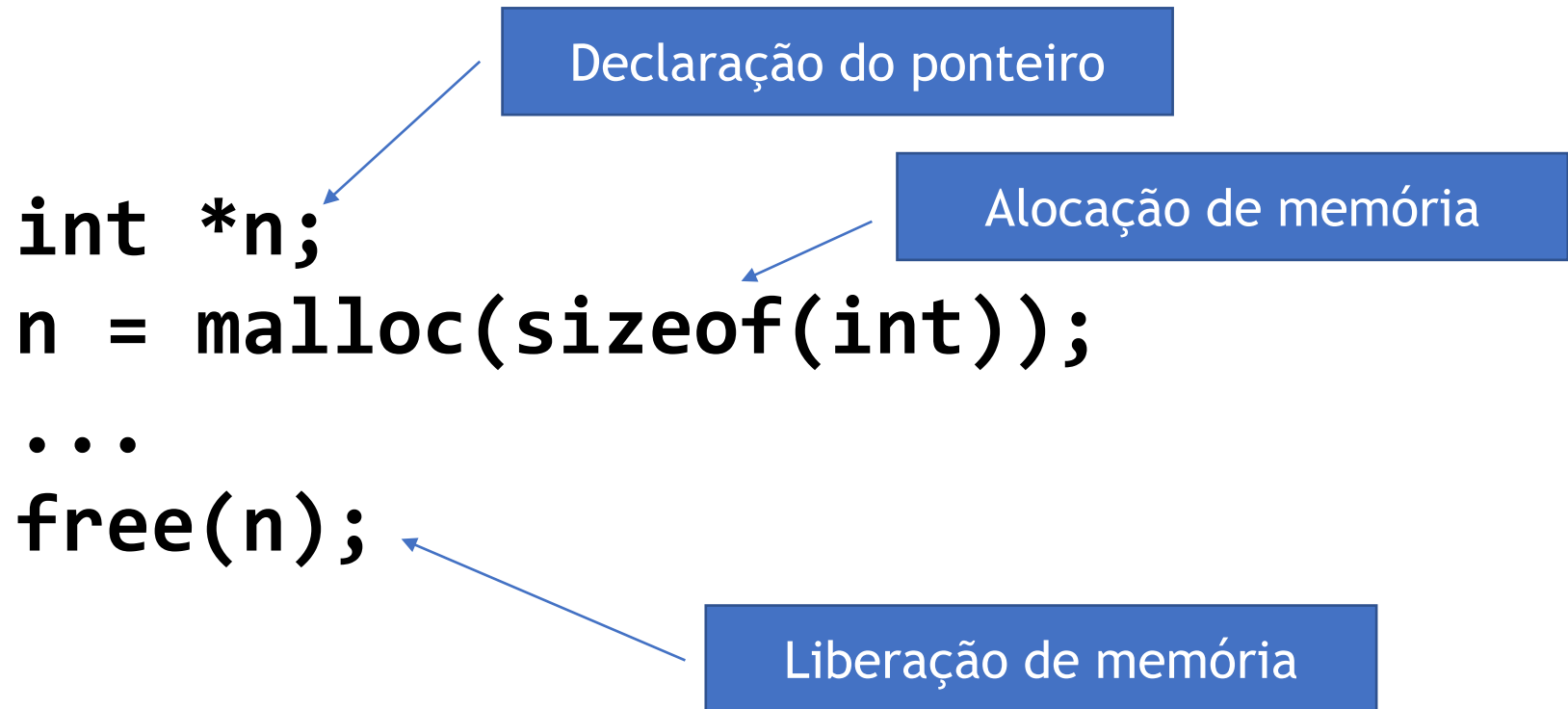
```
void* malloc( size_t size );
```

- 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?

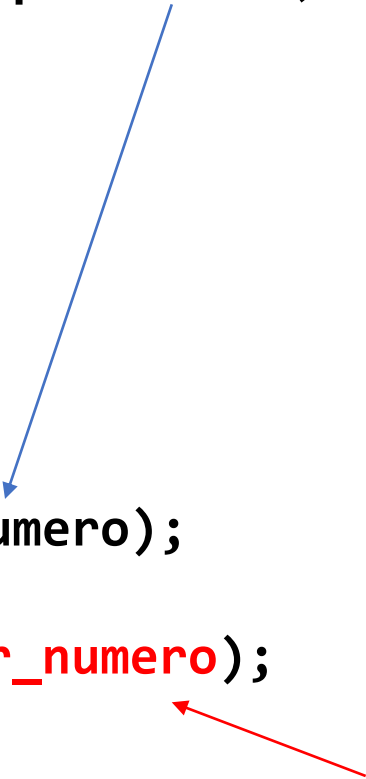
```
#include <stdio.h>
#include <stdlib.h>

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?

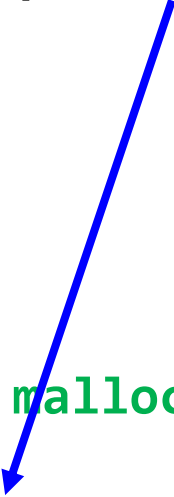
```
#include <stdio.h>
#include <stdlib.h>

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?

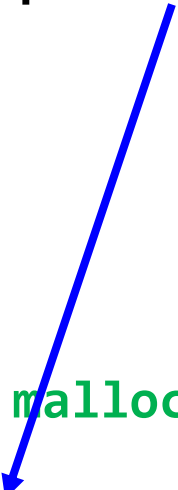
```
#include <stdio.h>
#include <stdlib.h>
```

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

    scanf("%d", ptr_numero);

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


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





# Exercícios

# Exercício 1

- Faça uma função que recebe um vetor de números e retorne:
  - Média
  - Desvio padrão   $\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \mu)^2}$
  - Menor e maior número
- Todos esses valores devem ser retornados com parâmetros passados por referência.

# Exercício 2

- Use a função do exercício anterior e chame com variáveis instanciadas dinamicamente (malloc);
- **Lembre-se de chamar `free` ao final!**

# 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/>
- CELES, W.; CERQUEIRA, R.; RANGEL, J. L.  
Introdução a Estruturas de Dados.  
Elsevier/Campus, 2004.

# 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.