

Programação 2 _ T2

Gestão de memória e apontadores. Alocação dinâmica de memória.

C ~~e Assembly~~.

Rui Camacho
(slides por Luís Teixeira)
MIEEC 2020/2021

VARIÁVEIS DINÂMICAS

As variáveis locais têm **dimensão fixa** e ocupam uma área de memória invariável durante a execução do programa. Do ponto de vista da memória, não é o programador que cria as variáveis; é o compilador que as cria.

Então, como criar e destruir variáveis, aumentar e diminuir a sua dimensão, durante a execução do programa?

As linguagens de programação permitem definir **variáveis dinâmicas** à custa de rotinas que reservam espaço para variáveis durante a execução de um programa e o libertam quando as variáveis já não são necessárias.

APONTADORES E VARIÁVEIS DINÂMICAS

Embora sejam muito usados na manipulação de vetores, os apontadores são especialmente úteis para lidar com esta gestão dinâmica da memória.

Neste caso o acesso do programador a um segmento de memória que tenha reservado é realizado **através de um apontador**.

A relação entre apontadores e vetores faculta a possibilidade de vermos um **segmento de memória como um vetor**.

APONTADORES – REVISÃO

Guardam uma referência, especificamente o **endereço** de uma variável (ou função) em memória

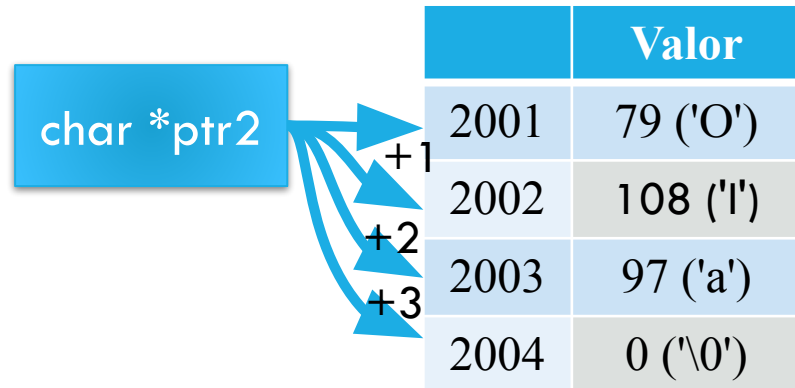
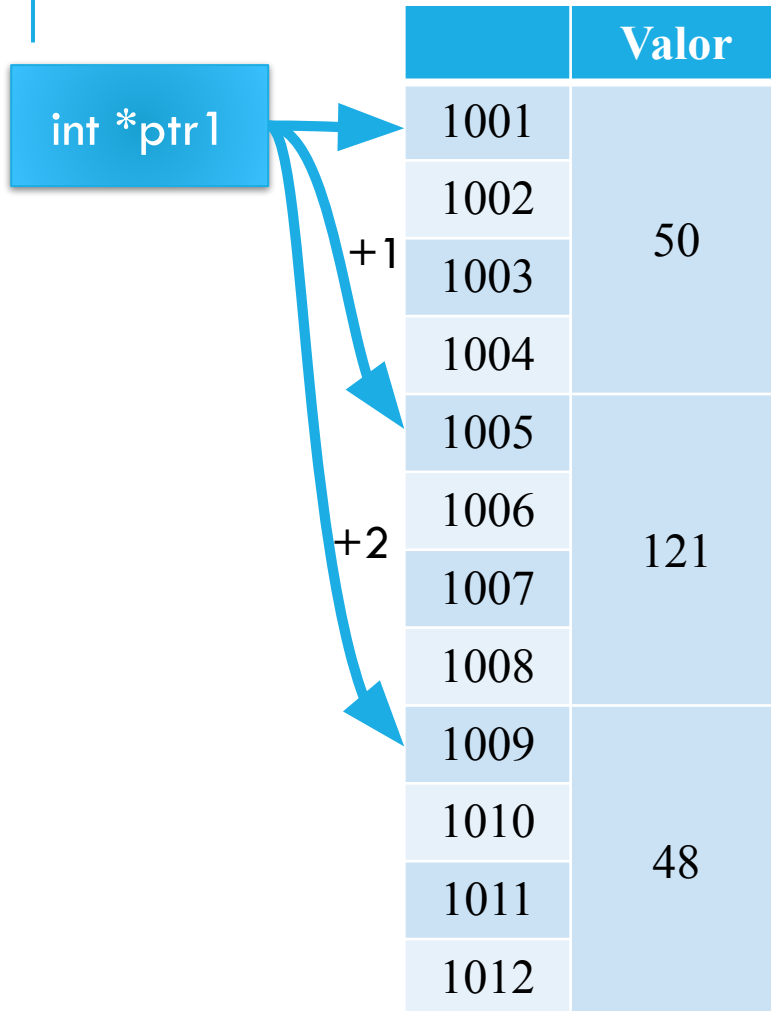
Podem ser “desreferenciados” para se aceder aos dados apontados

Podem ser manipulados utilizando atribuição ou aritmética de apontadores

Em C são utilizados tipicamente em:

- Vetores
- *Strings*
- Passagem de argumentos a funções por referência

ARITMÉTICA DE APONTADORES



Diferentes **tipos de apontadores** são necessários para determinar quantos bytes ocupa cada variável quando é usada **aritmética de apontadores**, por exemplo: `ptr++`

APONTADORES – EXEMPLO 1

```
#include <stdio.h>

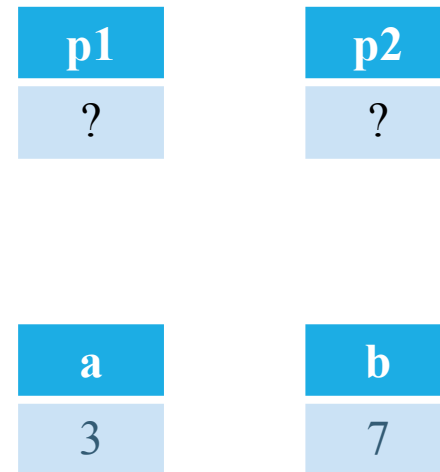
int main()
{
    int a = 3, b = 7;
    int *p1, *p2;
    p2 = p1 = &a;
    *p1 = 5;
    p2 = &b;
    printf("%d-%d", *p1, *p2);
}
```

O que imprime e qual o estado das variáveis em cada ponto de execução do código?

APONTADORES – EXEMPLO 1

```
#include <stdio.h>

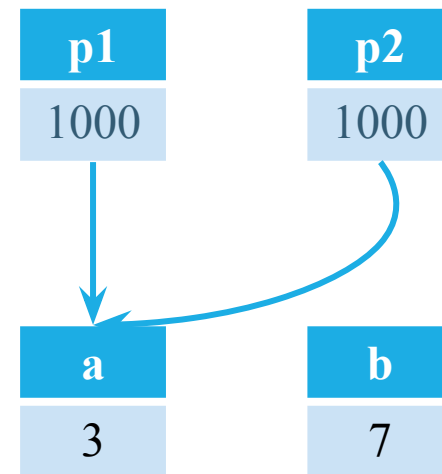
int main()
{
    int a = 3, b = 7;
    int *p1, *p2;
    p2 = p1 = &a;
    *p1 = 5;
    p2 = &b;
    printf("%d-%d", *p1, *p2);
}
```



APONTADORES – EXEMPLO 1

```
#include <stdio.h>

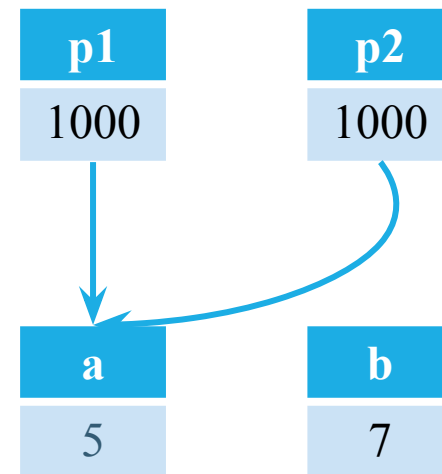
int main()
{
    int a = 3, b = 7;
    int *p1, *p2;
    p2 = p1 = &a;
    *p1 = 5;
    p2 = &b;
    printf("%d-%d", *p1, *p2);
}
```



APONTADORES – EXEMPLO 1

```
#include <stdio.h>

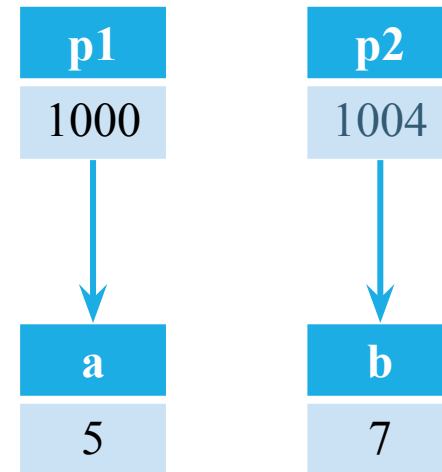
int main()
{
    int a = 3, b = 7;
    int *p1, *p2;
    p2 = p1 = &a;
    *p1 = 5;
    p2 = &b;
    printf("%d-%d", *p1, *p2);
}
```



APONTADORES – EXEMPLO 1

```
#include <stdio.h>

int main()
{
    int a = 3, b = 7;
    int *p1, *p2;
    p2 = p1 = &a;
    *p1 = 5;
    p2 = &b;
    printf("%d-%d", *p1, *p2);
}
```



APONTADORES – EXEMPLO 2

```
#include <stdio.h>

void func(int a, int *x)
{
    int s = a%2;
    a = (1-2*s)*a;
    *x = a*5;
}

int main()
{
    int a = 5, b = 7;
    func(a, &b);
    printf("%d-%d", a, b);
}
```

O que imprime e qual o estado das variáveis em cada ponto de execução do código?

APONTADORES – EXEMPLO 2

```
#include <stdio.h>

void func(int a, int *x)
{
    int s = a%2;
    a = (1-2*s)*a;
    *x = a*5;
}

int main()
{
    int a = 5, b = 7;
    func(a, &b);
    printf("%d-%d", a, b);
}
```

a
5

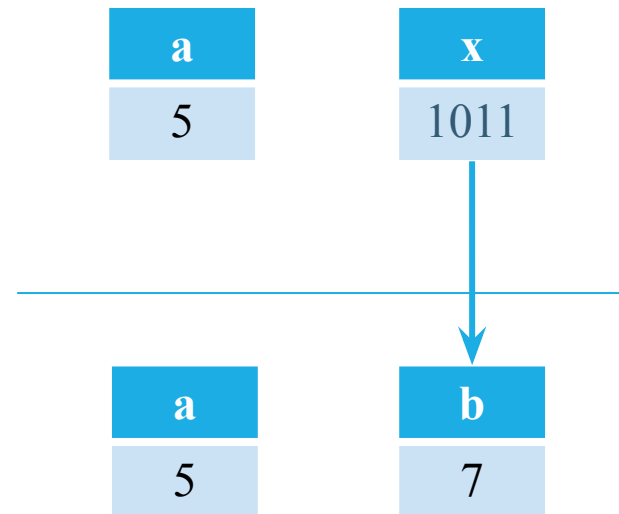
b
7

APONTADORES – EXEMPLO 2

```
#include <stdio.h>

void func(int a, int *x)
{
    int s = a%2;
    a = (1-2*s)*a;
    *x = a*5;
}

int main()
{
    int a = 5, b = 7;
    func(a, &b);
    printf("%d-%d", a, b);
}
```

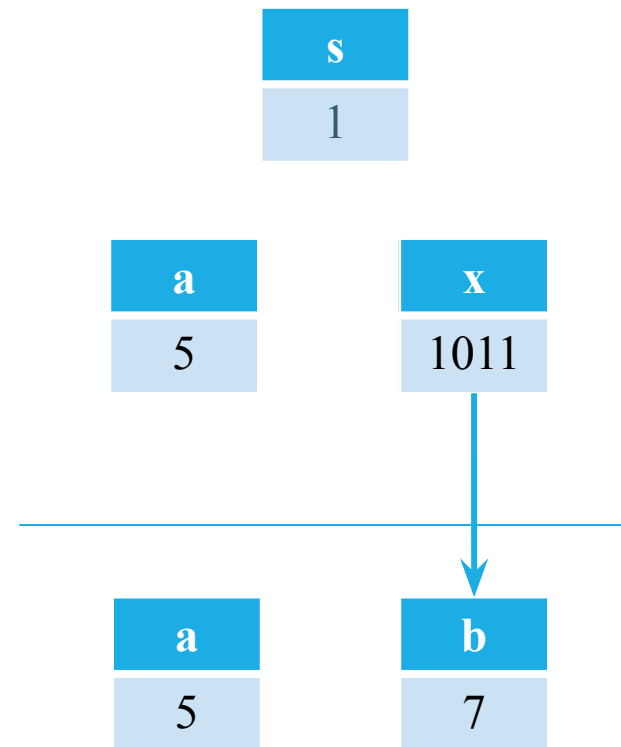


APONTADORES – EXEMPLO 2

```
#include <stdio.h>

void func(int a, int *x)
{
    int s = a%2;
    a = (1-2*s)*a;
    *x = a*5;
}

int main()
{
    int a = 5, b = 7;
    func(a, &b);
    printf("%d-%d", a, b);
}
```

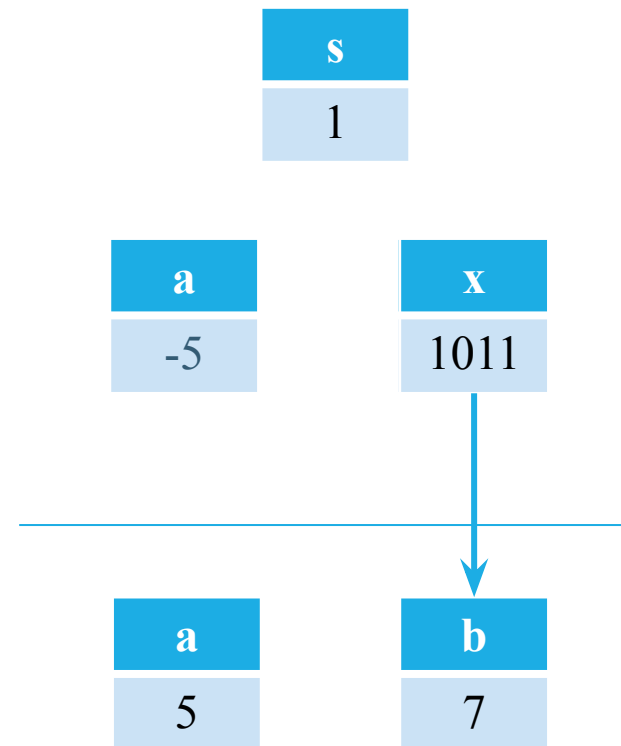


APONTADORES – EXEMPLO 2

```
#include <stdio.h>

void func(int a, int *x)
{
    int s = a%2;
    a = (1-2*s)*a;
    *x = a*5;
}

int main()
{
    int a = 5, b = 7;
    func(a, &b);
    printf("%d-%d", a, b);
}
```

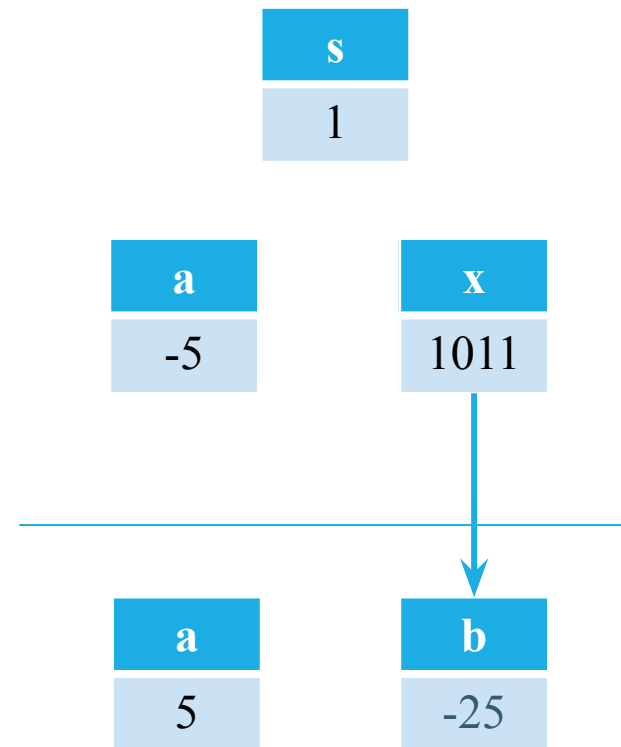


APONTADORES – EXEMPLO 2

```
#include <stdio.h>

void func(int a, int *x)
{
    int s = a%2;
    a = (1-2*s)*a;
    *x = a*5;
}

int main()
{
    int a = 5, b = 7;
    func(a, &b);
    printf("%d-%d", a, b);
}
```



APONTADORES – EXEMPLO 2

```
#include <stdio.h>

void func(int a, int *x)
{
    int s = a%2;
    a = (1-2*s)*a;
    *x = a*5;
}

int main()
{
    int a = 5, b = 7;
    func(a, &b);
    printf("%d-%d", a, b);
}
```

a
5

b
-25

APONTADORES – EM MEMÓRIA

Na realidade os apontadores são também variáveis

- São guardados em memória em conjunto com as restantes variáveis
- Têm um endereço associado
- Podem “apontar” por outro apontador



VARIÁVEIS DINÂMICAS

Gestão de memória dinâmica em C é feita através de um conjunto de funções na biblioteca *standard* `stdlib.h`

- ❑ `malloc/calloc`
- ❑ `free`
- ❑ `realloc`

Atribuição do espaço de memória durante a alocação e libertação de variáveis é feita por um algoritmo específico

ALOCAÇÃO: MALLOC

malloc retorna um apontador para um espaço de memória que podemos utilizar da forma que pretendemos

```
void *malloc(int size)
```

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

main()
{
    int *p;
    p = (int*)malloc(sizeof(int));
    *p = 10;
    printf("%d\n", *p);
}
```

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

main()
{
    int n;
    int *p;
    p = &n;
    *p = 10;
    printf("%d\n", *p);
}
```

OUTRO EXEMPLO

```
char *p = malloc(15);  
/* incompleto - valor de retorno do malloc nao verificado */  
strcpy(p, "Hello, world!");
```

```
char *somestring, *copy;  
...  
copy = malloc(strlen(somestring) + 1);    /* +1 for '\0' */  
/* incompleto - valor de retorno do malloc nao verificado */  
strcpy(copy, somestring);
```

```
int *ip = malloc(100 * sizeof(int));
```

```
if(ip == NULL) {  
    printf("Sem memoria\n");  
    exit or return  
}
```

Com teste do valor retornado

FREE

free liberta o espaço de memória apontado pelo apontador p

```
void free(void *p)
```

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

main()
{
    int *p;
    p = (int *) malloc(sizeof(int)); /* aloca espaço */
    *p = 10;
    printf("%d\n", *p);
    free(p); /* liberta espaço */
}
```

EVITAR ERROS

Não é obrigatório libertar a memória quando já não é necessária. No fim do programa, a memória pedida ao sistema operativo é automaticamente devolvida.

No caso de se pretender efectivamente libertar memória

```
free(p) ;  
p = NULL;
```

é uma **boa prática**, desde que associada com a prática de **testar o apontador** antes do seu uso.

Um espaço de memória reservado por uma função não é libertado quando se retorna ao programa principal: o apontador (variável local) deixa de existir, mas espaço de memória, não → deixa de ser possível libertar esse espaço.

VETORES DINÂMICOS

```
#include <stdio.h>

int* criavetor(int n)
/* Cria um vetor com n inteiros */
/* preenchido com os valores de 0 a n-1 */
{
    int *pi, i;
    pi = (int *) malloc(n*sizeof(int)); /* aloca espaço para n inteiros
*/
    for(i = 0; i < n; i++) pi[i] = i; /* preenche pi */
    return pi; /* retorna apontador */
}

int main()
{
    int *pi , i, n;
    printf("Quantos elementos tem o vector?"); scanf("%d",&n);
    pi = criavetor(n);
    for(i = 0; i < n; i++) printf("%d\n", pi[i]); /* ou *(pi+i) */
}
```


REALOCAÇÃO: *REALLOC*

realloc retorna um apontador para a realocação de um espaço de memória previamente alocado

```
void *realloc(void *p, int new_size)
```

```
#include <stdio.h>
#include <stdlib.h>
#define NCAR 80

main() {
    char *s;
    s = (char *) malloc(NCAR);
    printf("Escreva uma frase:");
    fgets(s, NCAR, stdin);
    printf("%s\n", s);
    s = (char *) realloc(s, 20);
    strcpy(s, "Bom dia! ");
    printf("%s\n", s);
}
```

CALLOC VS. MALLOC

calloc retorna um apontador para um espaço de memória que permite armazenar **nelements** de tamanho **size**

```
void *calloc(int nelements, int size);
```

```
/* aloca espaco para array de 10 elementos int */
int *ptr = calloc(10, sizeof (int));

if (ptr == NULL) {
    printf("Sem memoria\n");
    exit(1);
}
/* alocao bem sucedida */
```

malloc() não inicializa a memória alocada, enquanto que **calloc()** inicializa a memória alocada a ZERO

calloc(m, n) é o mesmo que
`p = malloc(m * n); for(i=0;i<m;i++) *(p+i)=0;`

GESTÃO DE MEMÓRIA

Em C a memória pode ser gerida de forma:

- Estática – variáveis estáticas são alocadas na memória principal e **persistem** para todo o ciclo de vida do programa

```
static int i = 0;
```

- Automática – variáveis alocadas na **stack**; são criadas e eliminadas quando as funções são chamadas e retornam; são variáveis válidas num contexto (scope) também designadas de **variáveis locais**

```
int i = 0;
```

- Dinâmica – variáveis são geridas explicitamente; são alocadas num espaço de memória livre designado **heap**

GESTÃO DE MEMÓRIA



Stack

- ❑ Memória alocada e libertada em last-in/first-out
- ❑ Alocação de memória consiste apenas em mudar posição de um apontador (*stack pointer*)
- ❑ Pequena, rápida, rígida

Heap

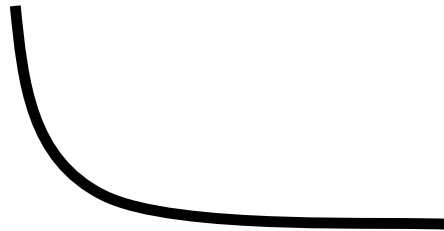
- ❑ Memória alocada quando necessário
- ❑ Exige manter um registo dos segmentos de memória usados
- ❑ Pode ficar fragmentada, ou seja, com segmentos livres sem utilização
- ❑ Grande, um pouco mais lenta, flexível

FRAGMENTAÇÃO DA HEAP



Tamanho da Heap: ~12MB

Memória usada: ~11MB



Legenda:

Cada bloco da imagem representa **4096** bytes em memória.

- Blocos pretos ☐ completamente usados
- Blocos brancos ☐ maioritariamente livres
- Blocos verdes ☐ completamente livres

Fonte: <http://pavlovdotnet.wordpress.com>



Tamanho da Heap: ~28MB

Memória usada: ~15MB