

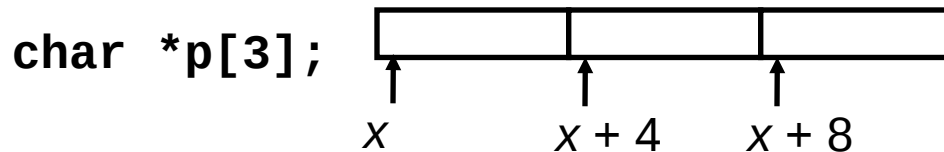
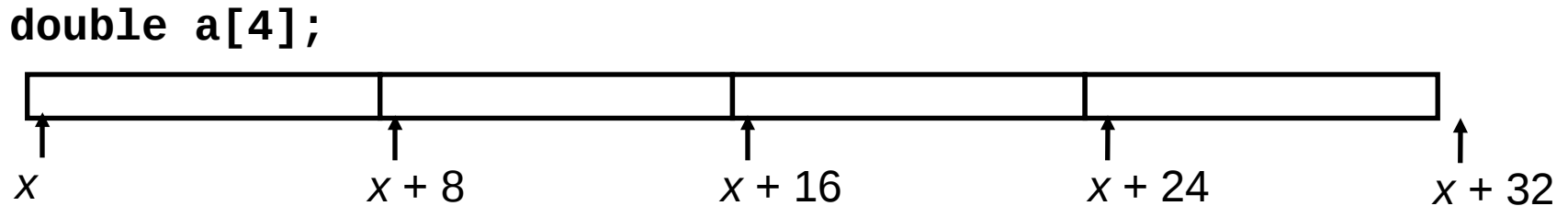
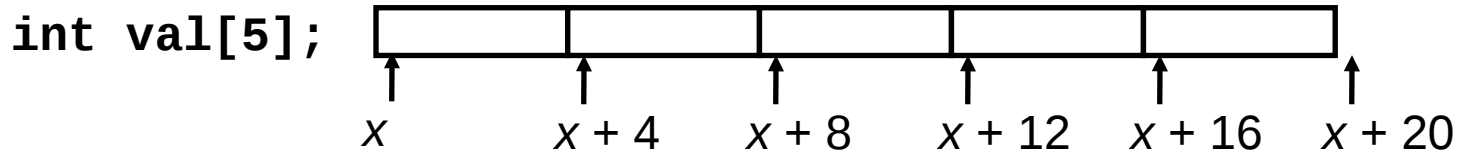
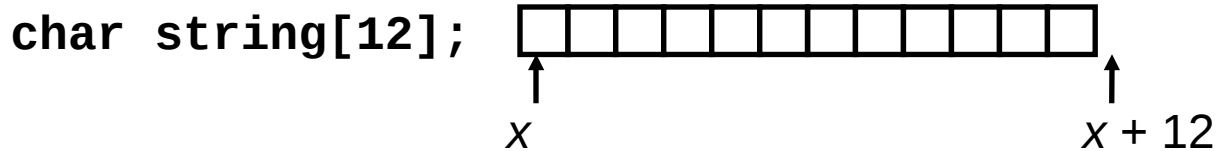
Representação de Dados

Arrays e Structs

Representação de arrays

- C usa uma implementação bastante simples de arrays
 - alocação contígua na memória
- Para um tipo **T** e uma constante **N**, a declaração **T a[N]** aloca uma região contígua de memória com **N X L** bytes, onde **L** é o tamanho em bytes do tipo **T** (`sizeof(T)`)
 - **`sizeof(a) = sizeof(T) * N`**
- Primeiro elemento (`a[0]`) corresponde ao menor endereço de memória

Alocação de arrays simples



Alocação de arrays: exemplos

```
char A[12];  
char *B[12];  
int C[12];  
double D[12];  
double *E[12];
```

Array	tam. do elemento	tamanho total	endereço inicial	end. do elemento i
A	1	12	X_A	$X_A + i$
B	4	48	X_B	$X_B + 4i$
C	4	48	X_C	$X_C + 4i$
D	8	96	X_D	$X_D + 8i$
E	4	48	X_E	$X_E + 4i$

Relação entre ponteiros e arrays

- O nome de um array equivale a um ponteiro para o tipo dos seus elementos
- Após a declaração **int a[tam]**
 - **a** é um ponteiro constante do tipo **int***, e seu valor é **&a[0]**
- Ao passarmos um nome de array como parâmetro estamos passando seu endereço (i.e., um ponteiro para seu primeiro elemento)

```
void formatArray(..., unsigned char *p, ...);  
...  
unsigned char seq[NBYTES];  
formatArray(..., seq, ...);
```

Aritmética básica com ponteiros

- C permite operações aritméticas com ponteiros
 - resultado é um endereço (valor de ponteiro)
 - resultado depende do tipo do dado referenciado pelo ponteiro
- $T * p \rightarrow (p + i)$ equivale a um deslocamento na memória de $p + \text{sizeof}(T) * i$
 - se declararmos `int *pi`, $(pi + 3)$ é um deslocamento de 12 bytes ($3 * \text{sizeof}(\text{int})$) a partir do endereço `pi`.
- Com aritmética de ponteiros, e equivalência entre arrays e ponteiros:

```
int a[5];  
int *p = a
```

} $a[3] \Leftrightarrow *(p + 3)$
} $p[3] \Leftrightarrow *(a + 3)$

Comparação de ponteiros

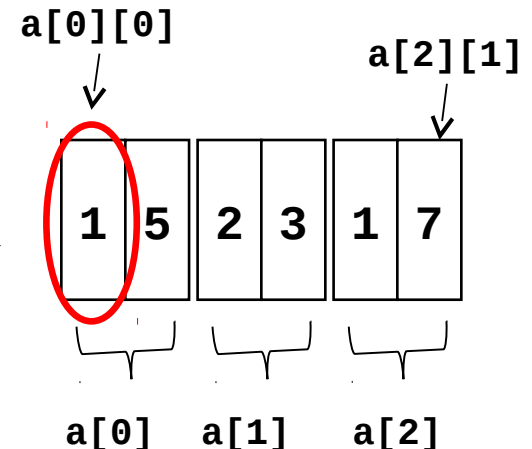
- Dois ponteiros podem ser comparados se referenciam um mesmo tipo
 - $p1 == p2$ se os dois ponteiros apontam **o mesmo elemento**
 - $p1 > p2$ se o valor (endereço contido em) de $p1$ é maior que o de $p2$
- $p1 - p2$ fornece o **número de elementos** entre $p1$ e $p2$
 - e **não** o número de bytes!
- Em tempo:
 - qual a diferença entre $*p++$ e $(*p)++$?

Arrays multi-dimensionais

- Mesma forma de alocação e acesso aos elementos
- `int a[3][2]` → **a** é um array de 3 elementos
 - cada elemento de **a** é um array (de 2 elementos)
 - **a** é um array de arrays → e **a** é um **ponteiro para um array**
 - **a[i]** é um array → e é um **ponteiro para inteiros**

```
int a[3][2] = {{1,5},{2,3},{1,7}};
```

int -> 4 bytes



Cálculo do endereço

```
int a[3][2] = {{1,5},{2,3},{1,7}};
```

- Endereço de $a[i] \rightarrow xa + i * \text{sizeof}(T)$

array de 2 inteiros

- Endereço do elemento $a[i][j]$

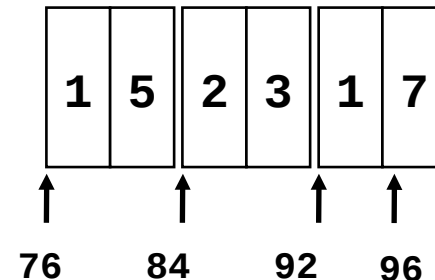
– Em bytes: $xa + i * 2 * \text{sizeof}(\text{int}) + j * \text{sizeof}(\text{int})$

$\underbrace{\hspace{10em}}_{\text{\&a[i]}}$

– Em C: $(\text{int} *)a + i * 2 + j$

$\underbrace{\hspace{10em}}_{\text{2 inteiros por array}}$

$a[2][1]$



$\text{\&a}[2][1] = 76 + (2*2*4) + 1*4 = 96$

Perguntas...

- Quando declaramos um parâmetro formal de um função C que é um array unidimensional, não precisamos declarar seu tamanho. Por que?
- Tanto faz declararmos um parâmetro formal como um array de inteiros ou como um ponteiro para inteiros. Por que?
- Se o parâmetro for um array bidimensional, teremos que declarar pelo menos o número de "colunas". Por que?

Estruturas Heterogêneas (structs)

- Os campos de uma struct são alocados na memória na ordem de declaração porém não necessariamente são contíguos
 - layout de structs na memória depende da questão de **alinhamento**
- Campos que correspondem a uma palavra devem ser alocados em endereços múltiplos de 4 (arquiteturas de 32 bits), campos do tipo short devem estar alinhados em endereços múltiplos de 2
 - os bytes não usados na memória alocada são denominados *padding*
- O início da estrutura deve estar alinhado
 - para que se possa “calcular” corretamente os endereços (offsets) de seus campos, respeitando as regras de alinhamento
 - isso é especialmente importante para arrays de *structs*, pois os elementos de um array são contíguos!

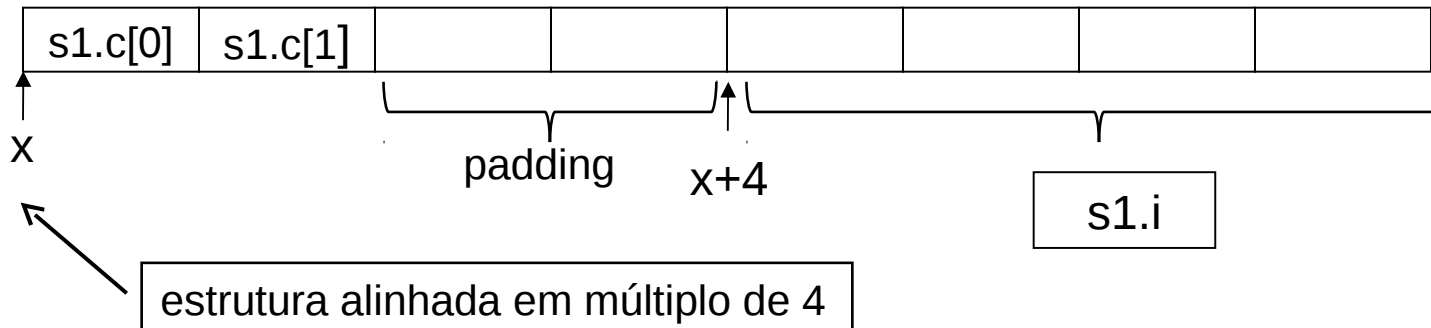
Exemplo de alinhamento

```
struct s{  
    char c[2];  
    int i;  
}  
struct s s1;
```

`sizeof(s1) = 8`

inteiro alinhado em múltiplo de 4

Representação na memória:

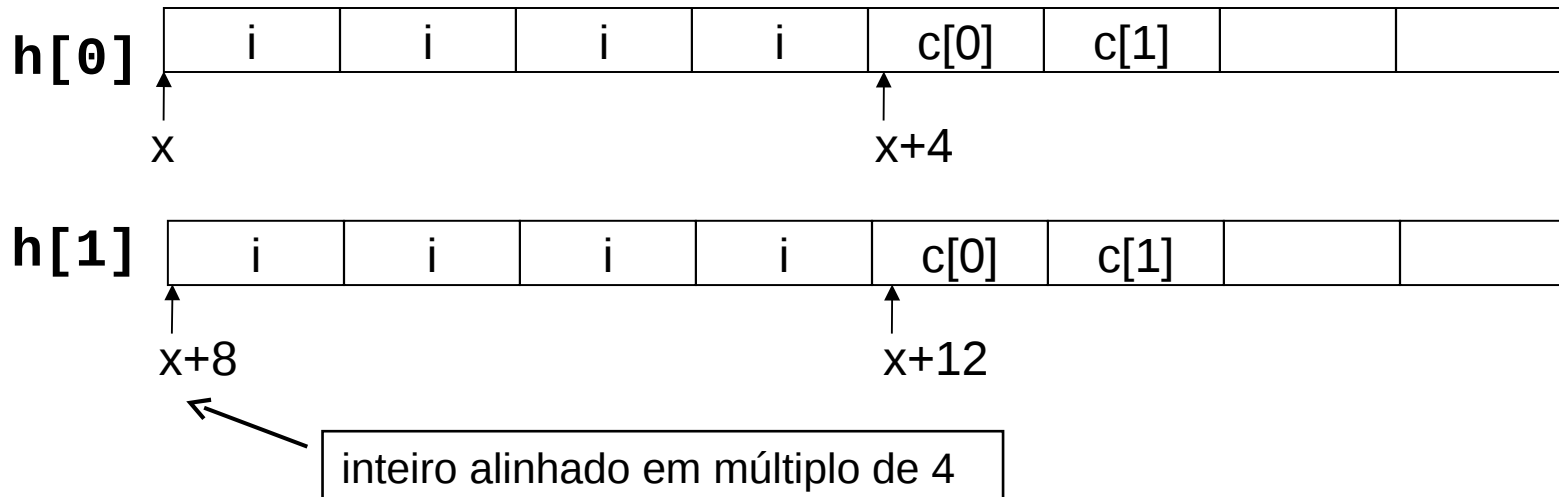


Padding no final

- Para garantir o alinhamento de *structs*, pode ocorrer um *padding* no final.

```
struct s{  
    int i;  
    char c[2];  
}  
struct s h[2];
```

`sizeof(h) = 16;`



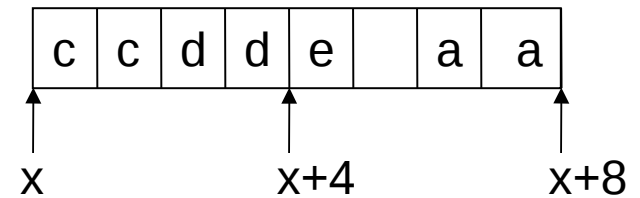
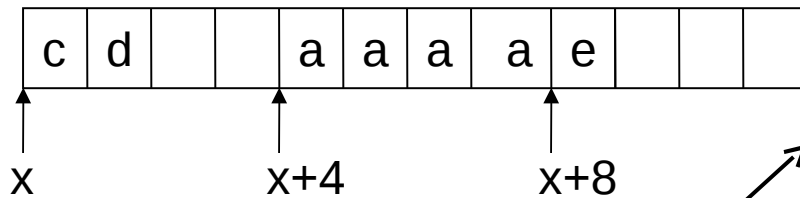
Exemplos de Alinhamento

```
struct s{  
    char c,d;  
    int a;  
    char e  
}  
struct s s1;
```

sizeof(s1) = 12

```
struct s{  
    short c,d;  
    char e;  
    short a  
}  
struct s s2;
```

sizeof(s2) = 8



alinhamento em múltiplo de 4