Representação de Dados
Arrays e Structs

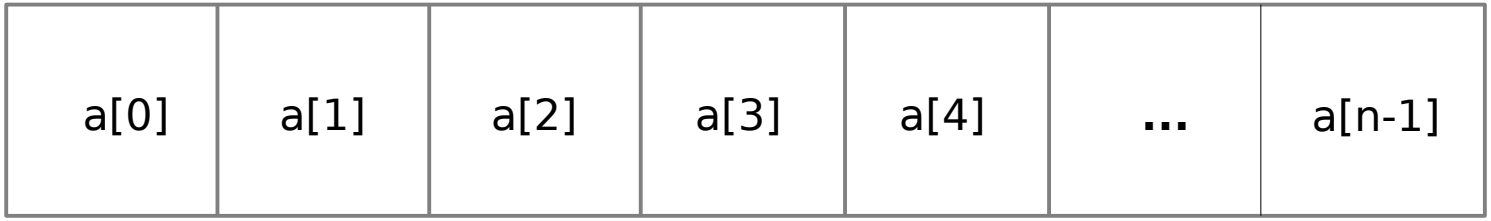
Noemi Rodriguez
Ana Lúcia de Moura
Raúl Renteria
Alexandre Meslin

http://www.inf.puc-rio.br/~inf1018

Representação de Arrays

C usa uma implementação bastante simples

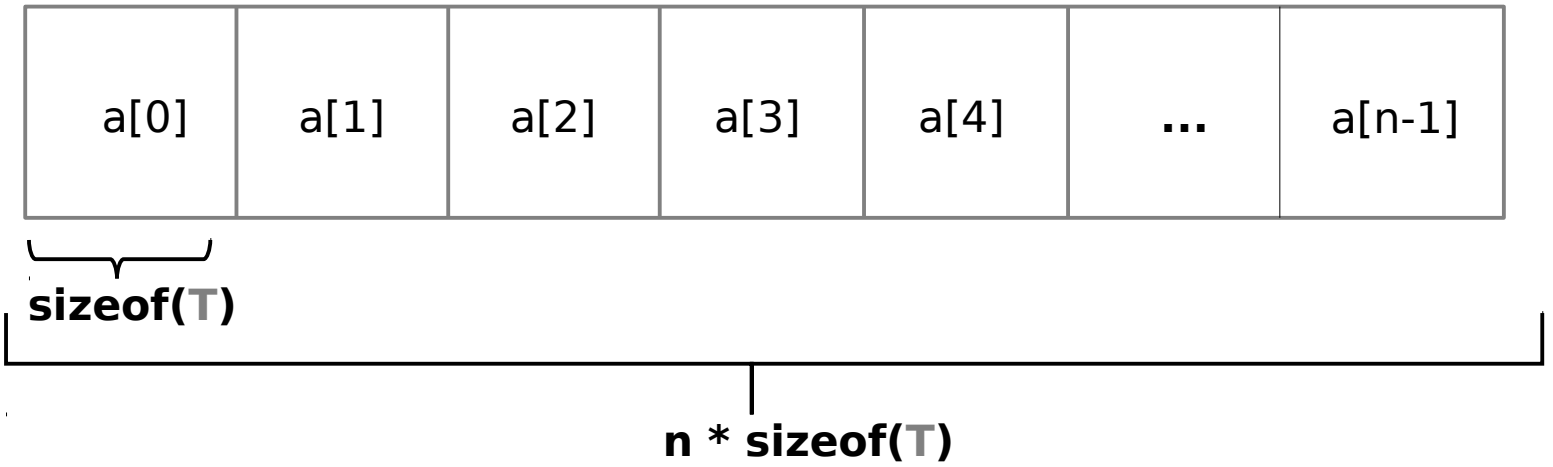
- alocação contígua na memória



Representação de Arrays

C usa uma implementação bastante simples

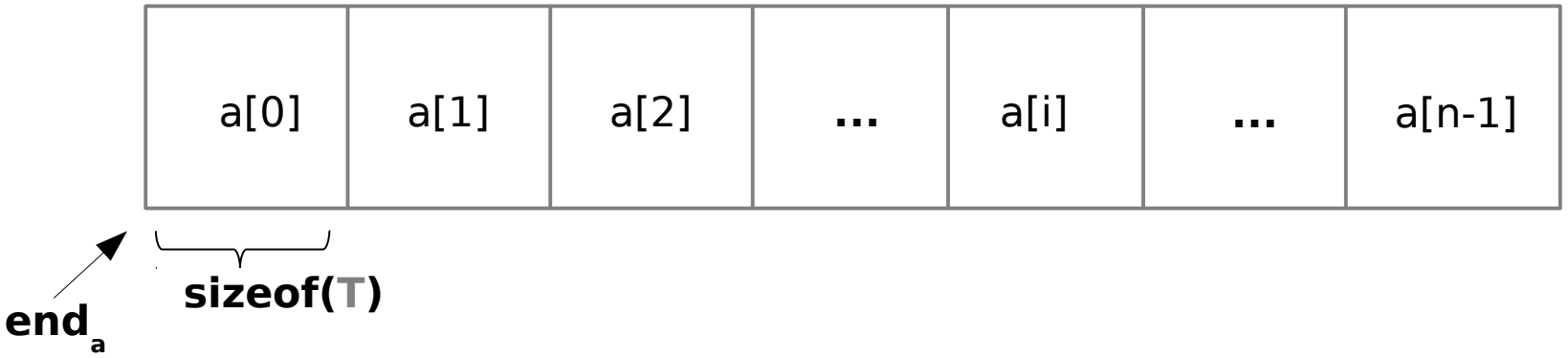
- alocação contígua na memória



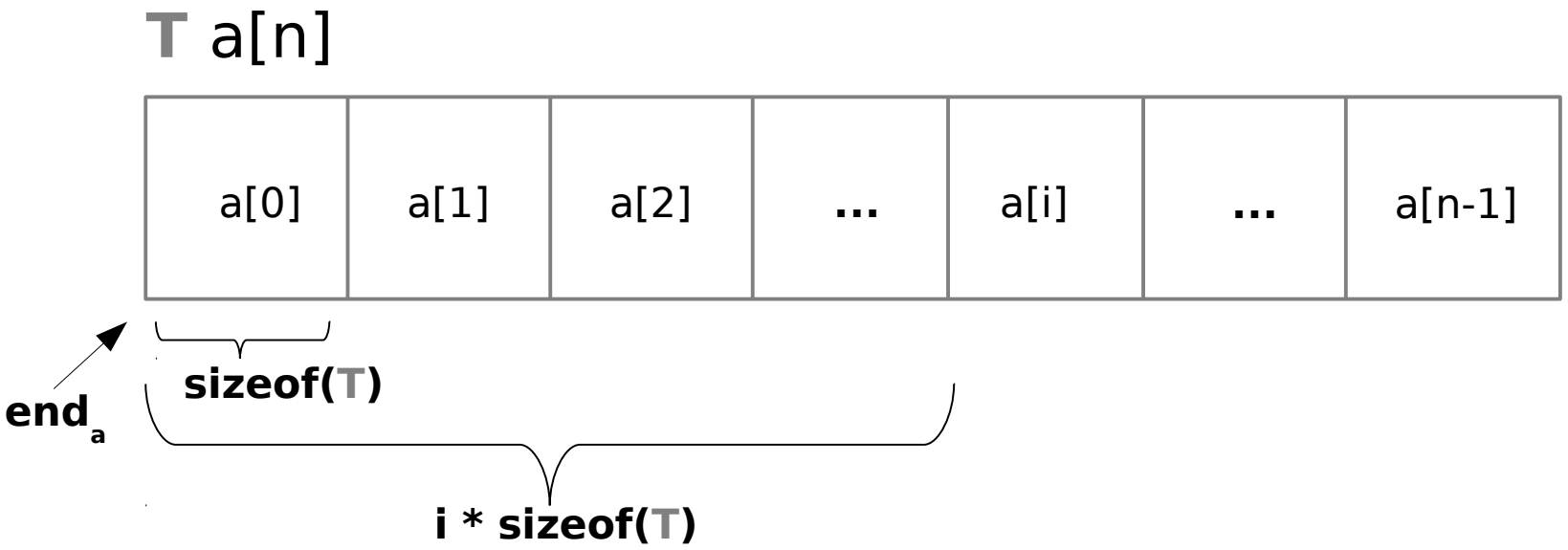
Para um tipo **T** e uma constante **n**, a declaração **T** a[**n**] aloca uma região contígua de memória com tamanho igual a **n** * **sizeof(T)** bytes

Endereços dos Elementos

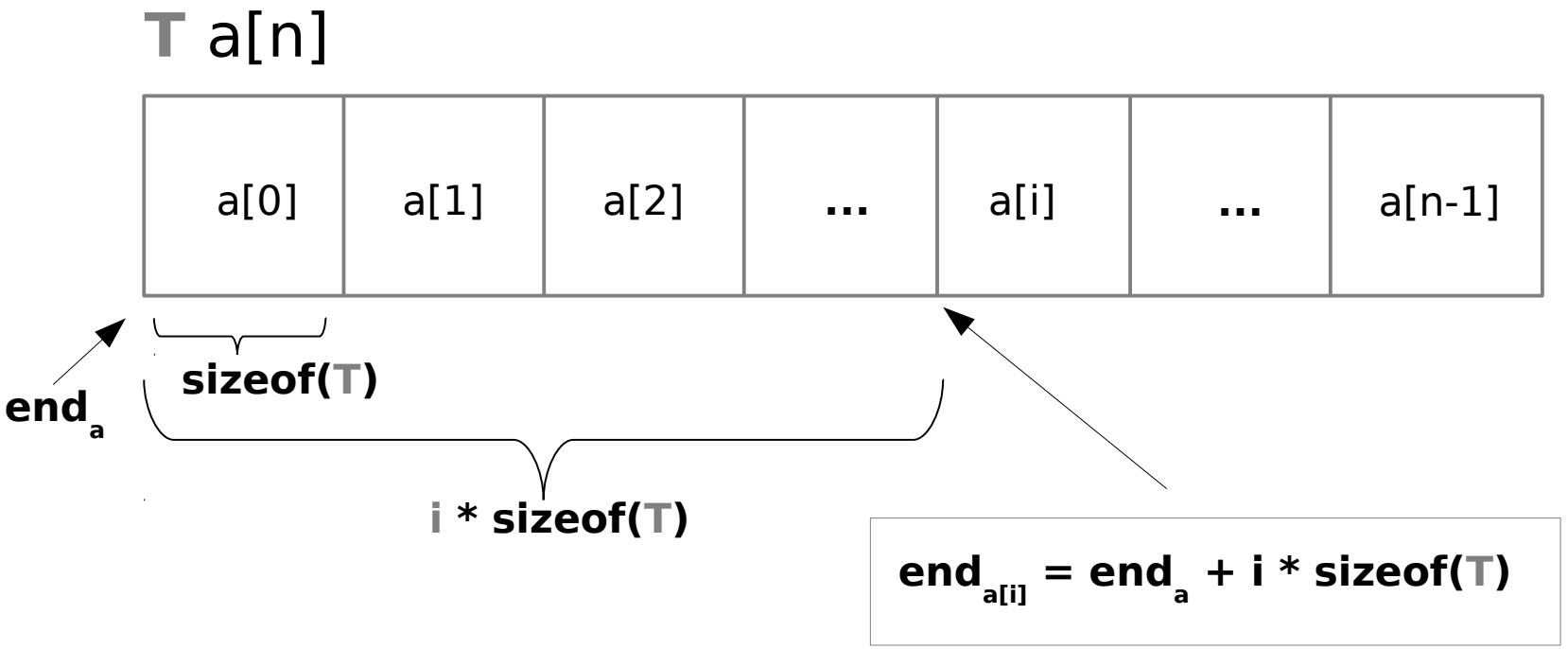
T a[n]



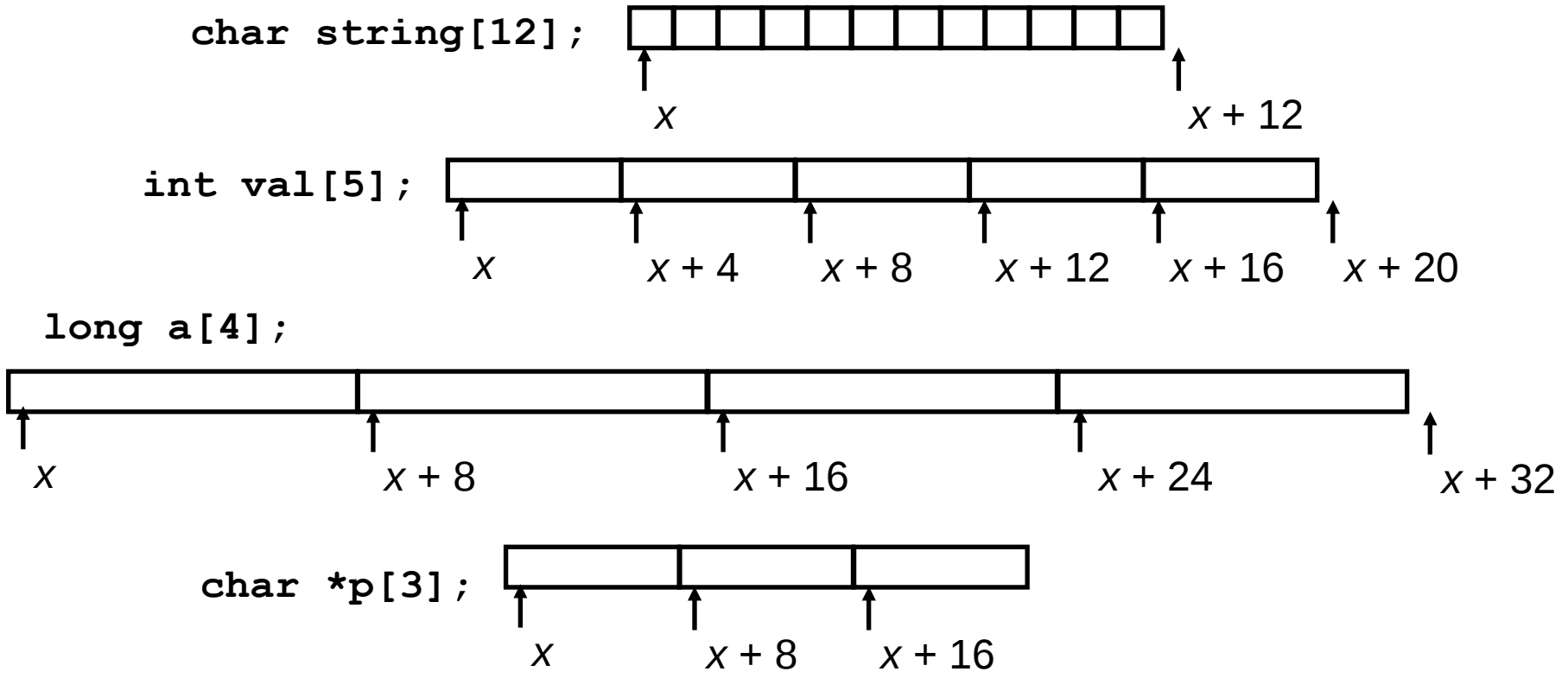
Endereços dos Elementos



Endereços dos Elementos



Alocação de Arrays Simples



Relação entre Ponteiros e Arrays

O nome de um array equivale a um ponteiro (constante) para o tipo de seus elementos

Após a declaração **int a[N]**

- **a** é um ponteiro constante do tipo **int** * e seu valor é **&a[0]**

Relação entre Ponteiros e Arrays

O nome de um array equivale a um ponteiro para o tipo de seus elementos

Após a declaração **int a[N]**

- **a** é um ponteiro constante do tipo **int** * e seu valor é **&a[0]**

Ao passarmos um array como parâmetro, passamos seu endereço (i.e., um ponteiro para seu primeiro elemento)

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

Aritmética com Ponteiros

C permite operações aritméticas básicas com ponteiros

- soma e subtração de valor inteiro: o resultado é um **endereço** e depende do **tipo de dado** referenciado pelo ponteiro

Aritmética com Ponteiros

C permite operações aritméticas básicas com ponteiros

- soma e subtração de valor inteiro: o resultado é um **endereço** e depende do **tipo de dado** referenciado pelo ponteiro

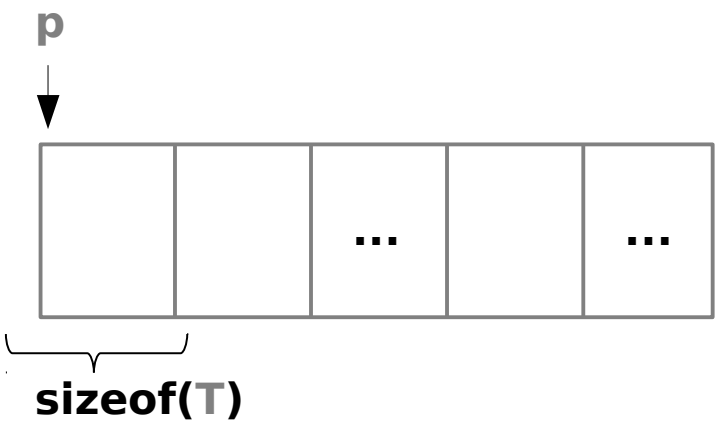
T *p → (p + i) equivale a um endereço na memória igual a **p + sizeof(T) * i**

Aritmética com Ponteiros

C permite operações aritméticas básicas com ponteiros

- soma e subtração de valor inteiro: o resultado é um **endereço** e depende do **tipo de dado** referenciado pelo ponteiro

T *p → (p + i) equivale a um endereço na memória igual a **p + sizeof(T) * i**

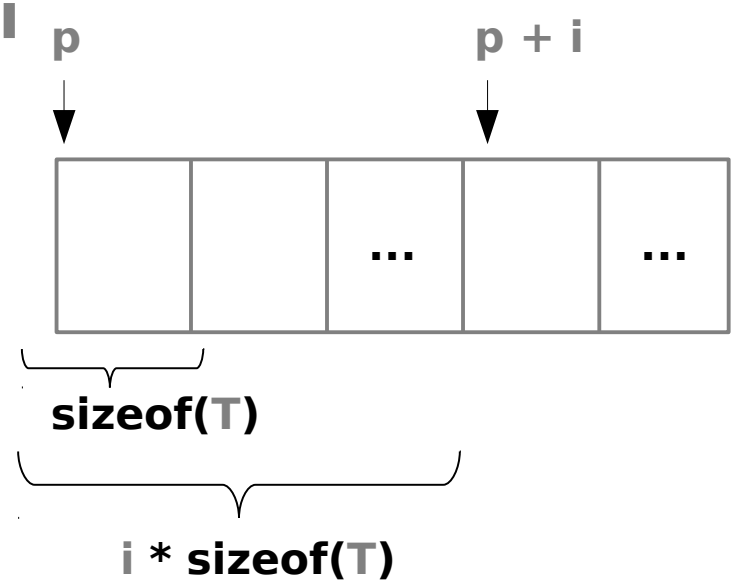


Aritmética com Ponteiros

C permite operações aritméticas básicas com ponteiros

- soma e subtração de valor inteiro: o resultado é um **endereço** e depende do **tipo de dado** referenciado pelo ponteiro

T *p → (p + i) equivale a um endereço na memória igual a **p + sizeof(T) * i**

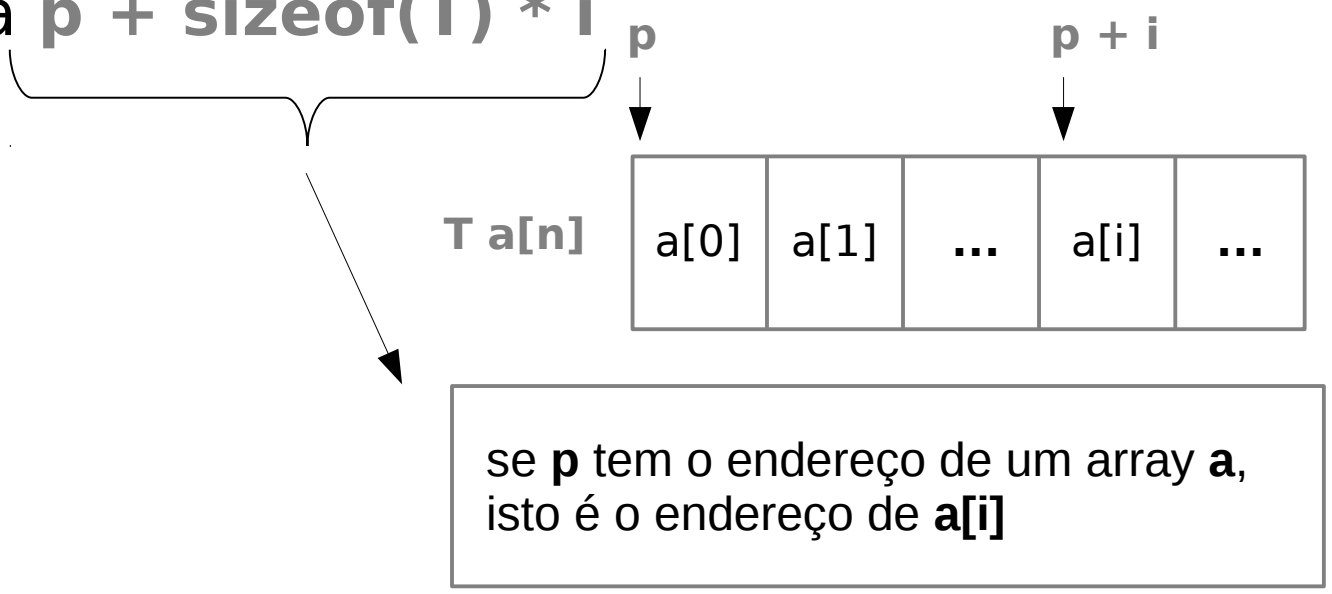


Aritmética com Ponteiros

C permite operações aritméticas básicas com ponteiros

- soma e subtração de valor inteiro: o resultado é um **endereço** e depende do **tipo de dado** referenciado pelo ponteiro

T *p → (p + i) equivale a um endereço na memória igual a **p + sizeof(T) * i**



Acessando Elementos de um Array

Com aritmética de ponteiros e relação entre ponteiros e arrays:

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

`a[3]` ⇔ `*(p + 3)`

Acessando Elementos de um Array

Com aritmética de ponteiros e relação entre ponteiros e arrays:

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

`a[3]` ⇔ `*(p + 3)`

`p[3]` ⇔ `*(a + 3)`

Arrays Multidimensionais

Mesma forma de alocação e acesso a elementos

elementos contíguos na memória

$end_{a[i]} = end_a + i * sizeof(T)$

Arrays Multidimensionais

Mesma forma de alocação e acesso a elementos

elementos contíguos na memória

$end_{a[i]} = end_a + i * sizeof(T)$

int a[3] [2]

Arrays Multidimensionais

Mesma forma de alocação e acesso a elementos

elementos contíguos na memória

$end_{a[i]} = end_a + i * sizeof(T)$

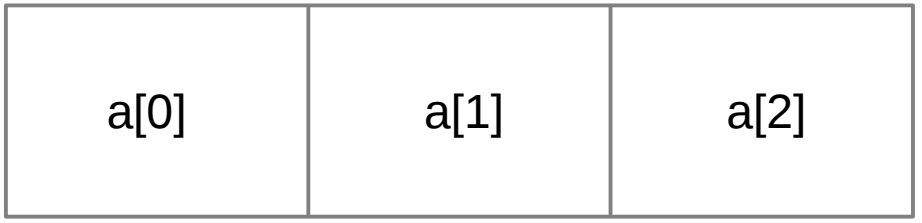
int a[3][2]

- **a** é um array de 3 elementos
  - **a[0]**, **a[1]** e **a[2]** são arrays de 2 inteiros

Exemplo

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

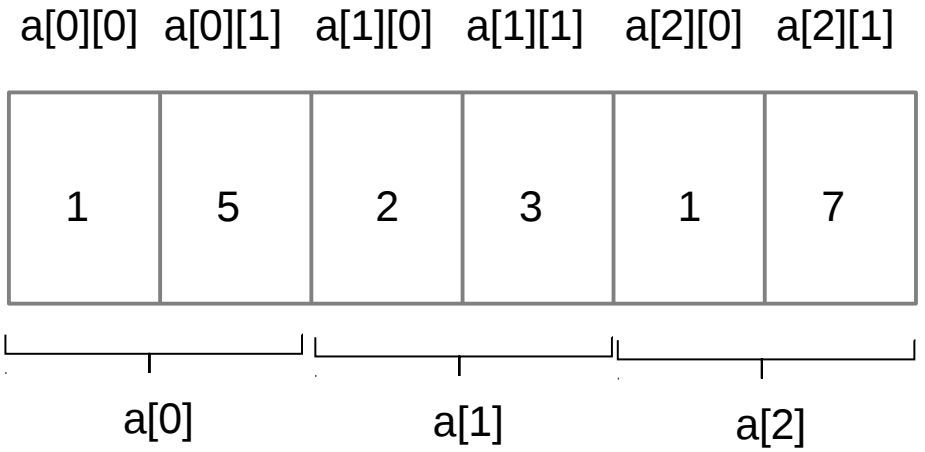
- **a** é um array de 3 elementos



Exemplo

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

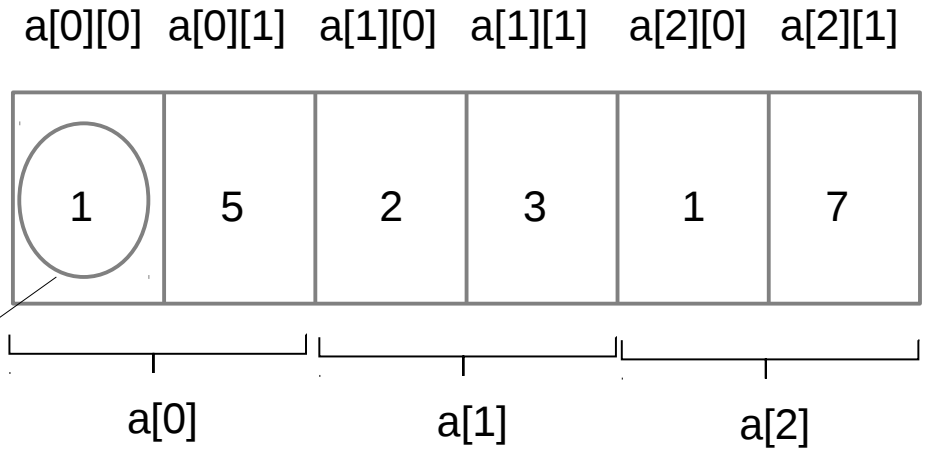
- **a** é um array de 3 elementos
- **a[0]**, **a[1]** e **a[2]** são arrays de 2 inteiros



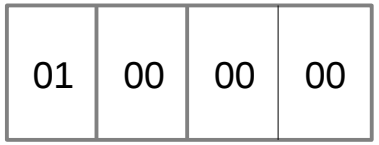
Exemplo

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

- **a** é um array de 3 elementos
- **a[0]**, **a[1]** e **a[2]** são arrays de 2 inteiros
- um inteiro tem 4 bytes



em little endian:

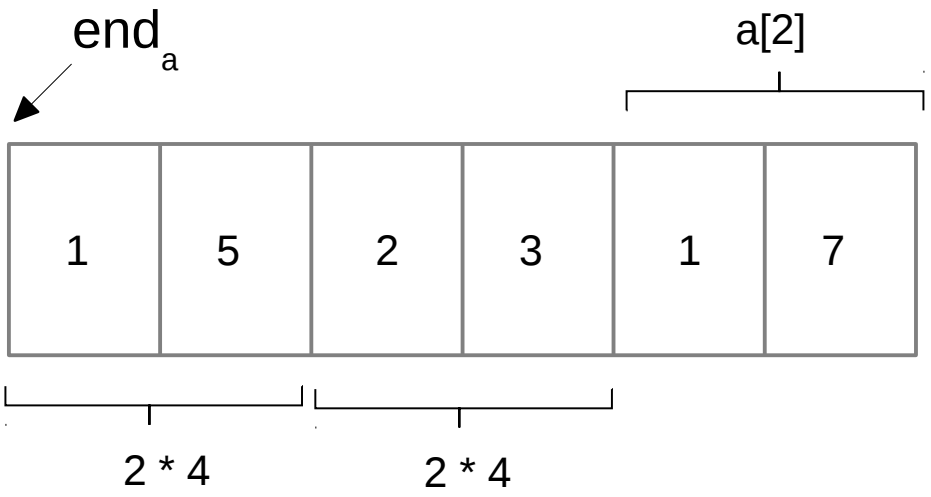


Cálculo do Endereço

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

- $\mathbf{end_{a[i]}} \rightarrow end_a + i * 2 * sizeof(int)$

sizeof(a[i])



Cálculo do Endereço

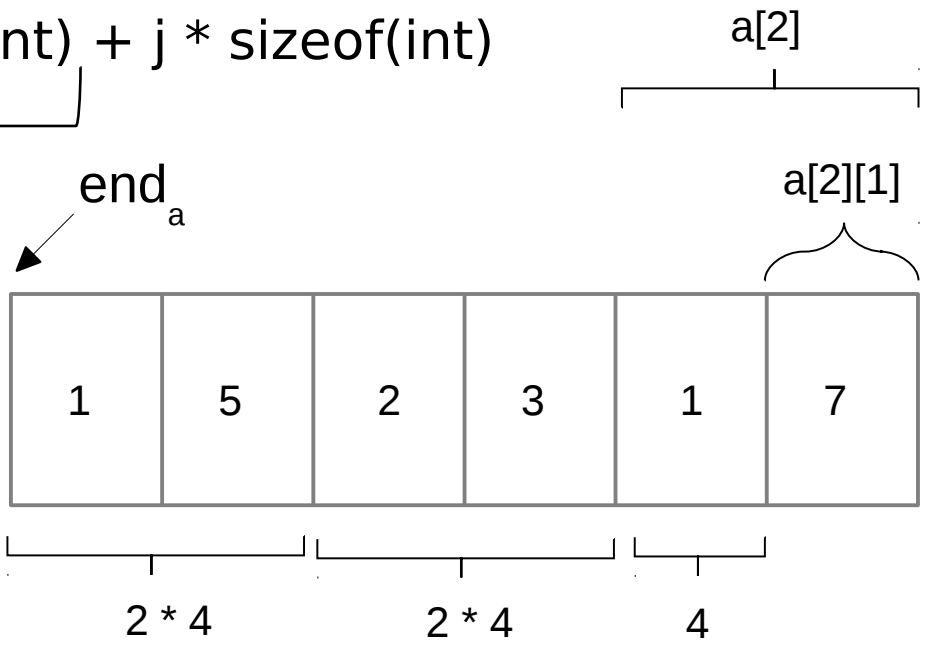
```
int a[3][2] = {{1,5},{2,3},{1,7};
```

- **end**$_{a[i]}$ → end$_a$ + i * 2 * sizeof(int)

sizeof(a[i])

- **end**$_{a[i][j]}$ → end$_a$ + i * 2 * sizeof(int) + j * sizeof(int)

end$_{a[i]}$



Perguntas

1. Quando declaramos um parâmetro formal de uma função C que é um array unidimensional, não precisamos declarar seu tamanho. Por que?

2. Mas se o parâmetro for um array bidimensional, teremos que declarar pelo menos o número de "colunas". Por que?

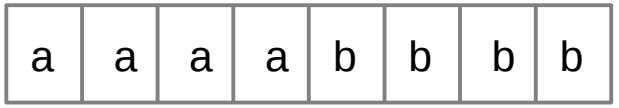
Estruturas Heterogênas (structs)

Os campos de uma struct são alocados na memória **sequencialmente**, porém **nem sempre contíguos**

Estruturas Heterogênas (structs)

Os campos de uma struct são alocados na memória **sequencialmente**, porém **nem sempre contíguos**

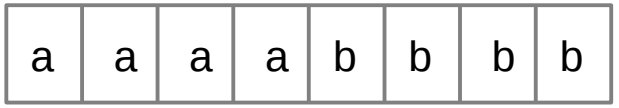
```
struct s {
    int a;
    int b;
}
struct s s1;
```



Estruturas Heterogênas (structs)

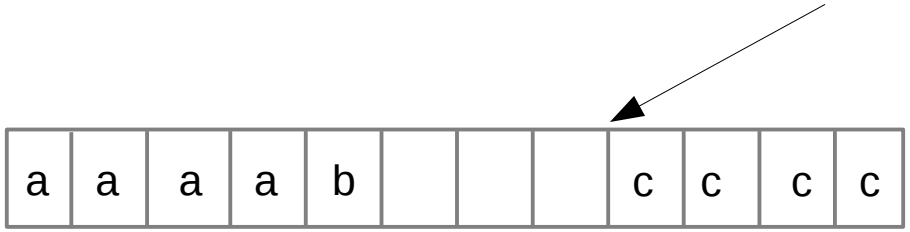
Os campos de uma struct são alocados na memória **sequencialmente**, porém **nem sempre contíguos**

```
struct s {
    int a;
    int b;
}
struct s s1;
```




```
struct s {
    int  a;
    char b;
   int c;
}
struct s s1;
```

layout de struct na memória depende da questão de **alinhamento**



DEPARTAMENTO DE INFORMÁTICA
PUC·RIO

Alinhamento de Dados

Em plataformas de 64 bits, dados escalares de tamanho igual a **k** bytes devem ser alocados em endereços múltiplos de **k**

- longs e ponteiros em **múltiplos de 8**, ints em **múltiplos de 4**, shorts em **múltiplos de 2**

Alinhamento de Dados

Em plataformas de 64 bits, dados escalares de tamanho igual a **k** bytes devem ser alocados em endereços múltiplos de **k**

- longs e ponteiros em **múltiplos de 8**, ints em **múltiplos de 4**, shorts em **múltiplos de 2**

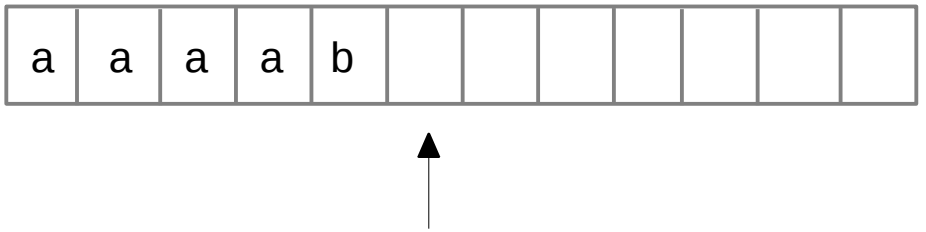
```
struct s {
    int  a;
    char b;
   int  c;
}
struct s s1;
```

Alinhamento de Dados

Em plataformas de 64 bits, dados escalares de tamanho igual a **k** bytes devem ser alocados em endereços múltiplos de **k**

- longs e ponteiros em **múltiplos de 8**, ints em **múltiplos de 4**, shorts em **múltiplos de 2**

```
struct s {
    int  a;
    char b;
   int  c;
}
struct s s1;
```



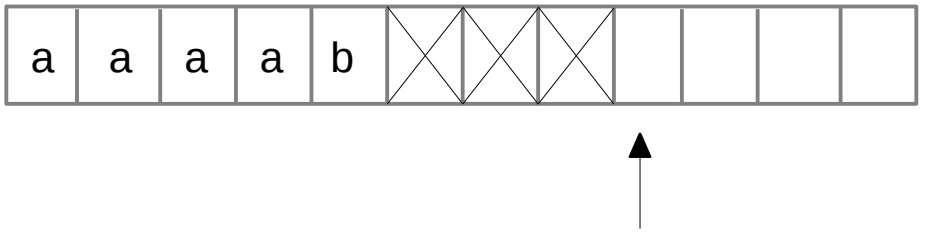
endereço múltiplo de 4 ?

Alinhamento de Dados

Em plataformas de 64 bits, dados escalares de tamanho igual a **k** bytes devem ser alocados em endereços múltiplos de **k**

- longs e ponteiros em **múltiplos de 8**, ints em **múltiplos de 4**, shorts em **múltiplos de 2**

```
struct s {
    int  a;
    char b;
   int  c;
}
struct s s1;
```



endereço múltiplo de 4

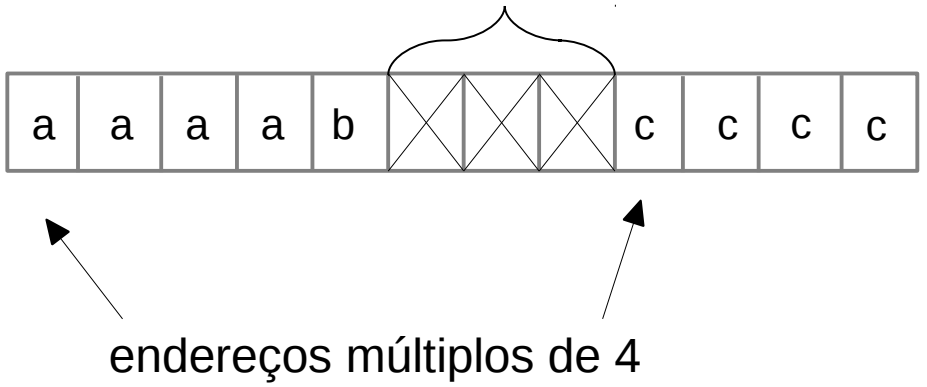
Alinhamento de Dados

Em plataformas de 64 bits, dados escalares de tamanho igual a **k** bytes devem ser alocados em endereços múltiplos de **k**

- longs e ponteiros em **múltiplos de 8**, ints em **múltiplos de 4**, shorts em **múltiplos de 2**

```
struct s {
    int  a;
    char b;
   int  c;
}
struct s s1;
```

bytes não usados são denominados **padding**



DEPARTAMENTO DE INFORMÁTICA
PUC·RIO

Padding no Final da Estrutura

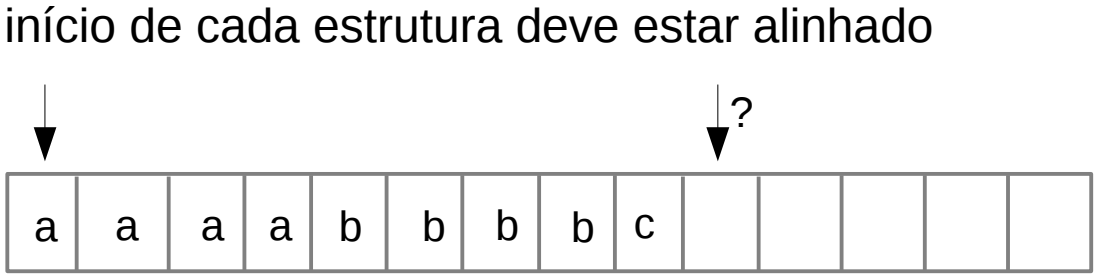
O tamanho (sizeof) de uma estrutura deve garantir o alinhamento para um array de estruturas

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

Padding no Final da Estrutura

O tamanho (sizeof) de uma estrutura deve garantir o alinhamento em um array de estruturas

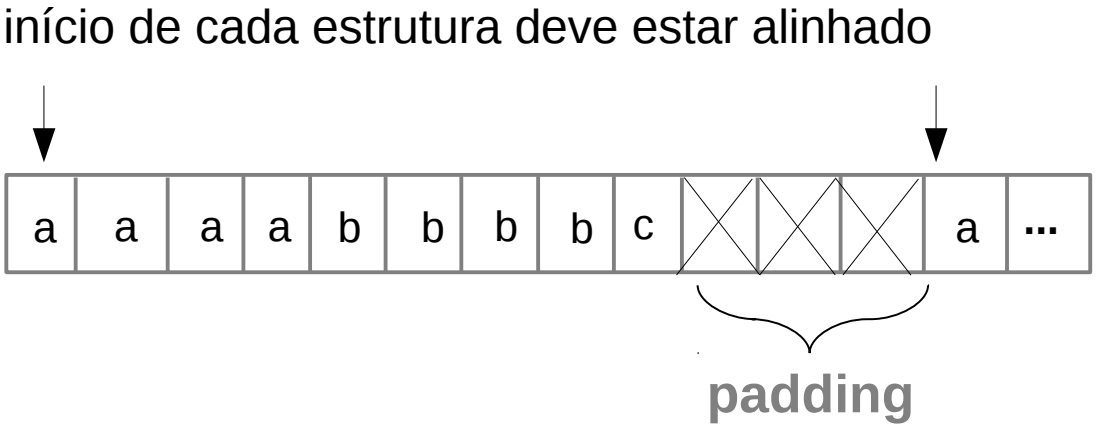
```
struct s {
   int  a;
    int  b;
   char c;
}
struct s s1[2];
```



Padding no Final da Estrutura

O tamanho (sizeof) de uma estrutura deve garantir o alinhamento em um array de estruturas

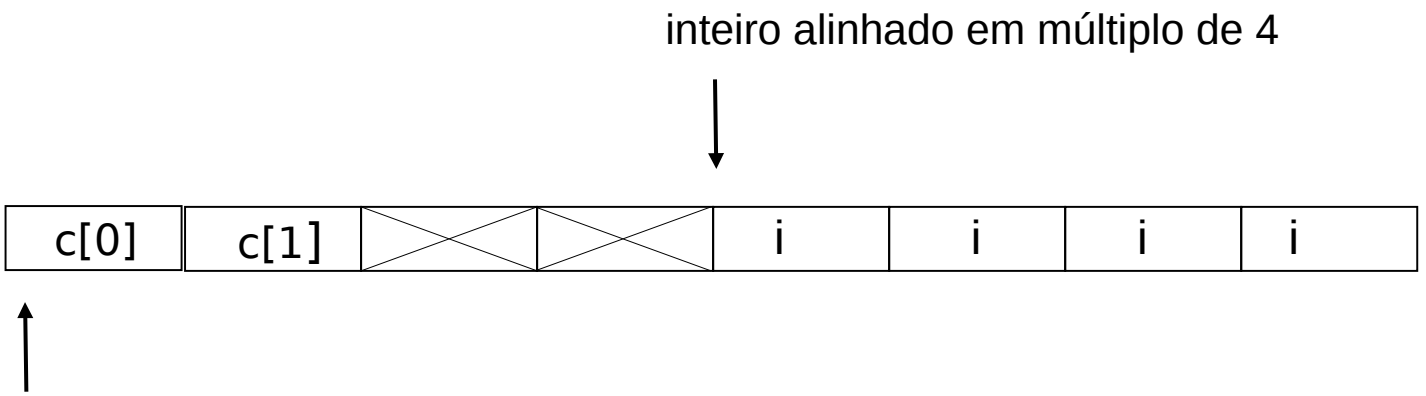
```
struct s {
   int  a;
   int  b;
   char c;
}
struct s s1[2];
```



Exemplos de Alinhamento

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

```
sizeof(s1) = 8
```



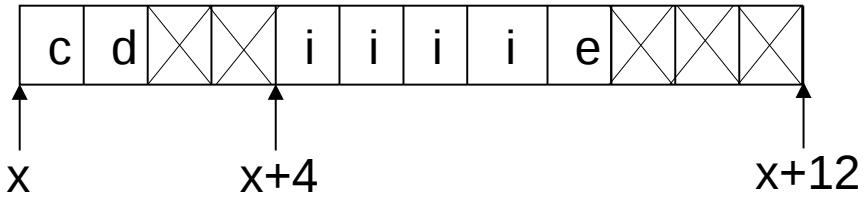
Exemplos de Alinhamento

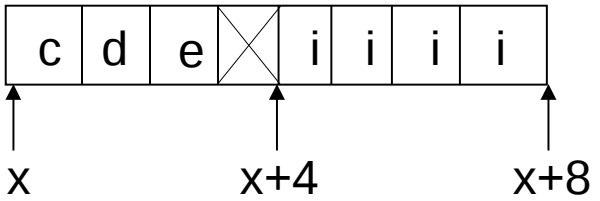
```
struct s {
    char c,d;
    int i;
    char e;
}
struct s s1;
```

```
struct t {
    char c,d;
    char e;
    int i;
}
struct t s2;
```

sizeof(s1) = 12

sizeof(s2) = 8



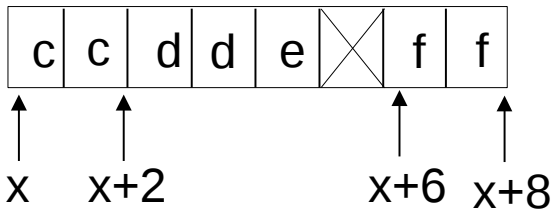


Exemplos de Alinhamento

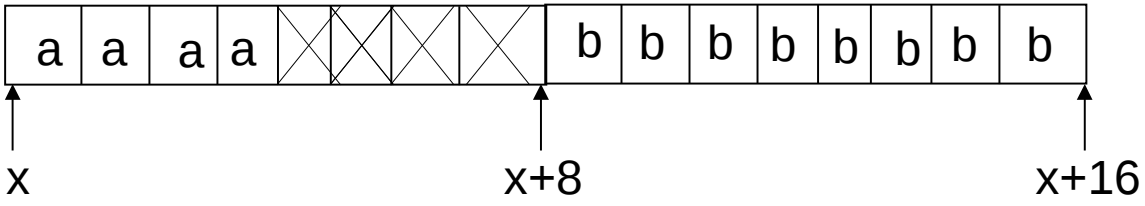
```
struct s {
    short c,d;
    char e;
    short f;
}
struct s s1;
```

```
struct t {
    int  a;
    long b;
}
struct t s2;
```

sizeof(s1) = 8



sizeof(s2) = 16



Unions

Unions permitem que um único “objeto” na memória seja referenciado por múltiplos tipos

A sintaxe da declaração é semelhante à de structs, porém a semântica é diferente

- uma **union** contém apenas UM de seus componentes em um dado momento

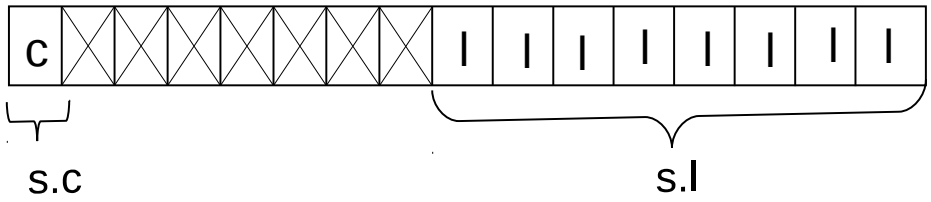
Unions

Unions permitem que um único “objeto” na memória seja referenciado por múltiplos tipos

A sintaxe da declaração é semelhante à de structs, porém a semântica é diferente

- uma **union** contém apenas UM de seus componentes em um dado momento

```
struct S {
    char c;
    long l;
} s;
```



sizeof(s) = 16

Unions

Unions permitem que um único “objeto” na memória seja referenciado por múltiplos tipos

A sintaxe da declaração é semelhante à de structs, porém a semântica é diferente

- uma **union** contém apenas UM de seus componentes em um dado momento

```
struct S {
    char c;
    long l;
} s;
```

```
union U {
    char c;
    long l;
} u;
```