Software Básico

Representação de Dados: Array, Estrutura e União

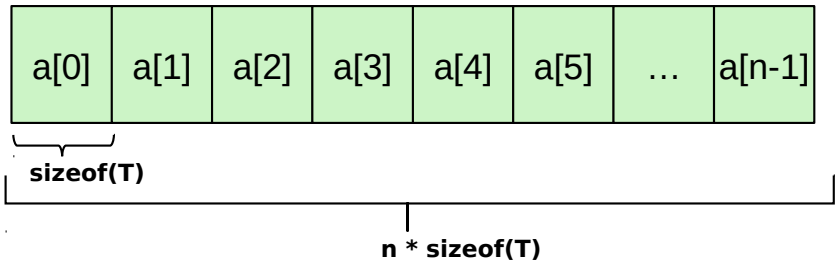
Reconhecimento

- Material produzido por:
  - Noemi Rodriguez – PUC-Rio
  - Ana Lúcia de Moura – PUC-Rio
- Adaptação
  - Bruno Silvestre – UFG

Array

Representação de Arrays

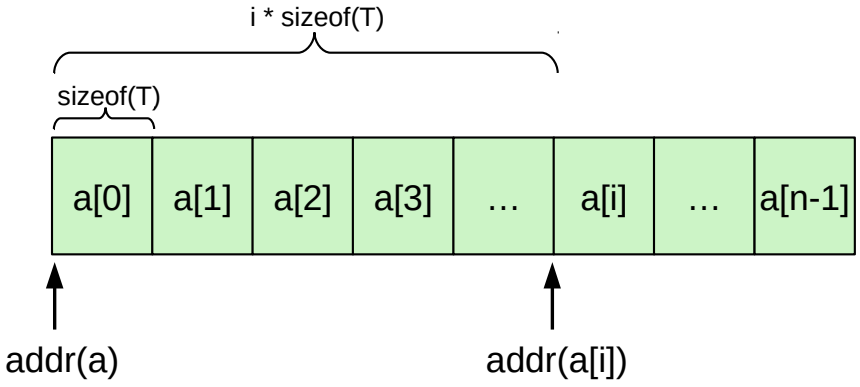
- C usa uma implementação bastante simples
  - alocação contígua na memória



- 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

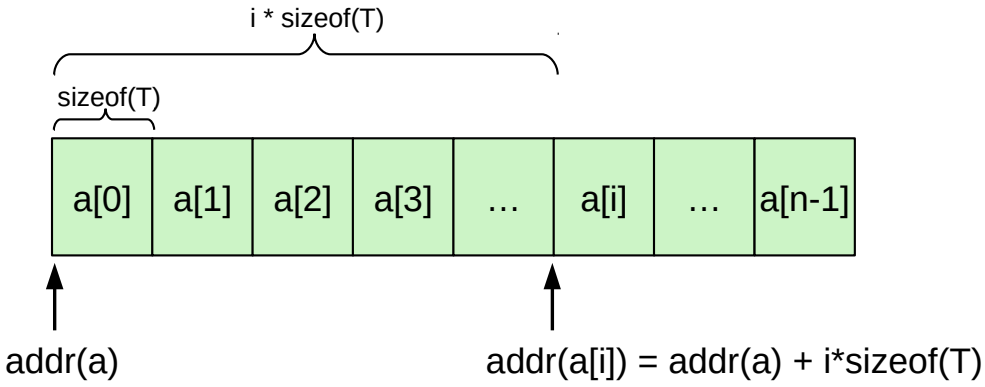
- Seja um vetor: T a[n];



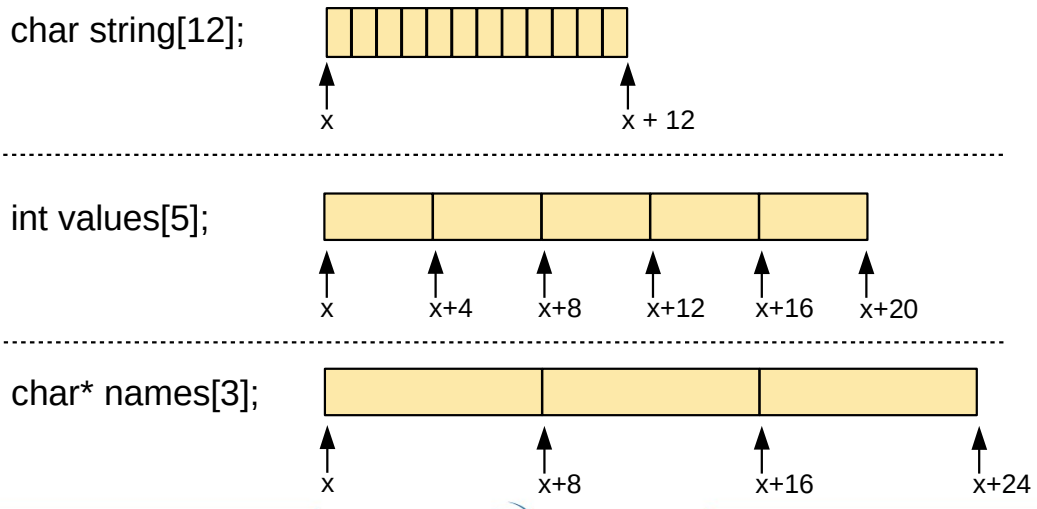
INSTITUTO DE INFORMÁTICA - UFG

Endereços dos Elementos

- Seja um vetor: T a[n];

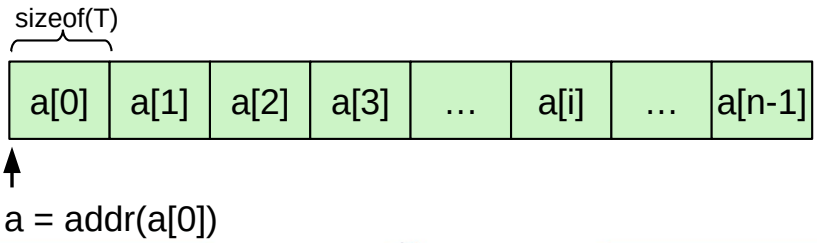


Alocação de Arrays Simples



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[5]”
  - “a” é um ponteiro constante do tipo “int *” e seu valor é &a[0]



Relação entre Ponteiros e Arrays

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

```
void sort(int *nums, int len)
{

  // do something

}
```

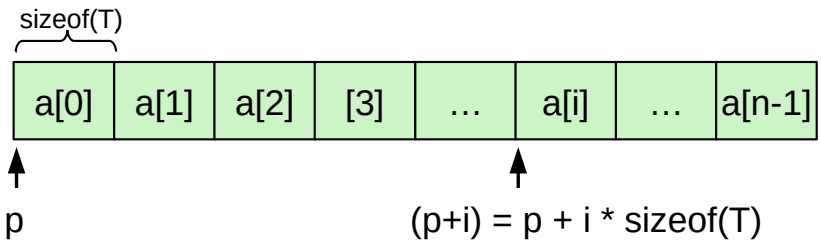
```
int numbers[16];

int main() {
  sort(numbers, 16);

  ...
}
```

Aritmética de Ponteiros

- C permite operações aritméticas básicas com ponteiros
  - Soma e subtração de valor inteiro com ponteiro
  - O resultado é um endereço e depende do tipo de dado referenciado pelo ponteiro

Aritmética de Ponteiros

- Seja a definição do ponteiro “T *p”
  - (p + i) equivale a um endereço na memória igual a p + i * sizeof(T)



Acessando Elementos de um Array

- Asterisco faz a dereferenciação do ponteiro

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

```
a[3] == *(p + 3)
```

Acessando Elementos de um Array

- Asterisco faz a dereferenciação do ponteiro
- Podemos usar notação "[ ]" com ponteiros também

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

```
a[3] == *(p + 3)
*(a + 3) == p[3]
```

Array Multidimensional

- C armazena na memória é linear
- Endereço do array multidimensional em ordem de linha, com índice começando em zero
  - Exemplo: int m[3][3]

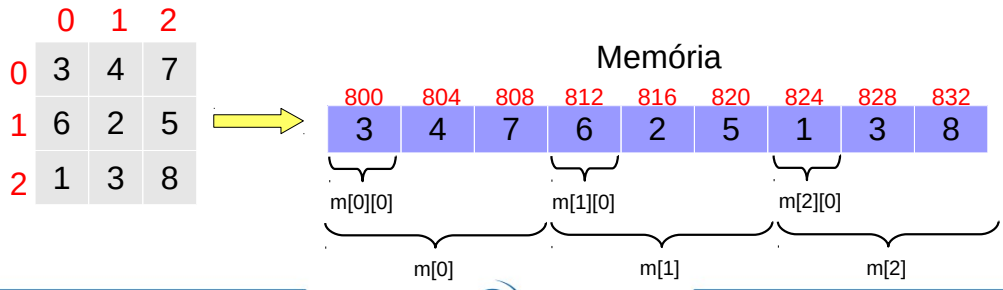
	0	1	2
0	3	4	7
1	6	2	5
2	1	3	8

Memória

800	804	808	812	816	820	824	828	832
3	4	7	6	2	5	1	3	8

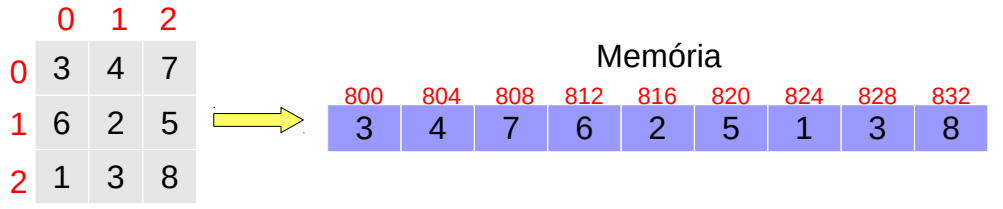
Array Multidimensional

- C armazena na memória é linear
- Endereço do array multidimensional em ordem de linha, com índice começando em zero
  - Exemplo: int m[3][3]



Array Multidimensional

```
addr(m[i][j]) = addr(m[0][0])          +
                (i * ncols * sizeof(T)) +
                (j * sizeof(T))
```



Estrutura

Estrutura

- Coleção de valores com tipos diferentes
- Elementos de uma *struct* geralmente são armazenados em sequência na memória
  - Mas nem sempre de forma contígua

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

struct s s1;
```

s1

a	a	a	a	b	b	b	b

Estrutura

- Coleção de valores com tipos diferentes
- Elementos de uma struct geralmente são armazenados em sequência na memória
  - Mas nem sempre de forma contígua

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

struct s S2;
```

S2

Alinhamento de Memória

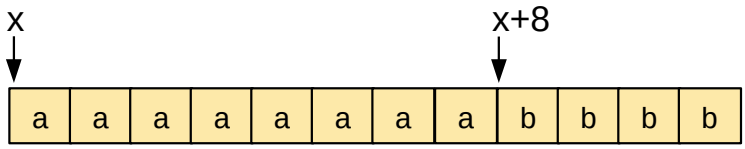
- Alguns processadores forçam um alinhamento de memória
- Em plataformas de 64 bits, dados escalares de tamanho igual a **k** bytes devem ser alocados em endereços múltiplos de **k**

TIPO	ENDEREÇO MÚLTIPLO DE
char	1
short	2
int	4
long	8
Ponteiro	8

Alinhamento de Memória

TIPO	ENDEREÇO MÚLTIPLO DE
char	1
short	2
int	4
long	8
Ponteiro	8

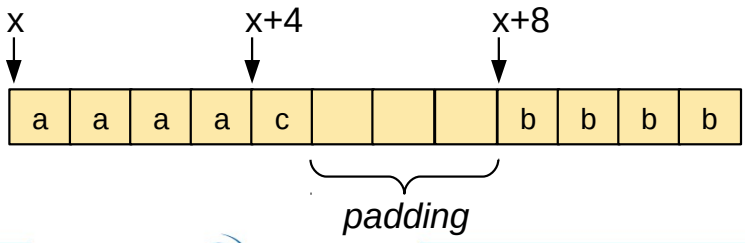
```
struct s {
  long  a;
  int   b;
};
```



Alinhamento de Memória

TIPO	ENDEREÇO MÚLTIPLO DE
char	1
short	2
int	4
long	8
Ponteiro	8

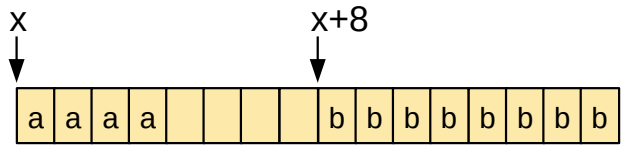
```
struct s {
  int   a;
  char c;
  int   b;
};
```



Alinhamento de Memória

- Endereço inicial também tem que estar alinhado

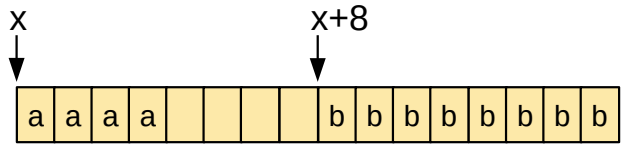
```
struct s {
  int   a;
  long  b;
};
```



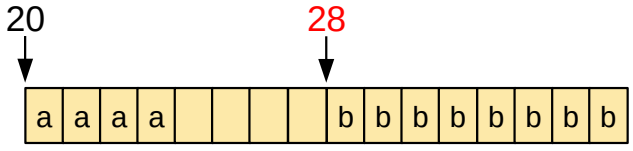
Alinhamento de Memória

- Endereço inicial também tem que estar alinhado

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

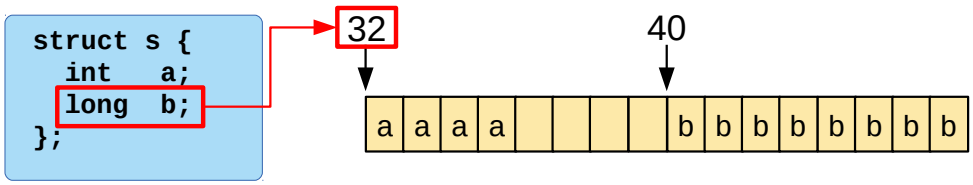


Se o endereço inicial da estrutura for “20”, “a” estará alinhado, mas “b” não!



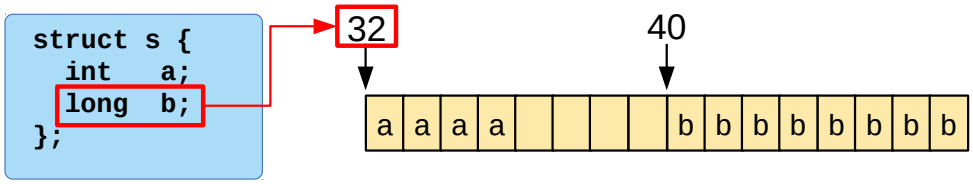
Alinhamento de Memória

- Endereço inicial também tem que estar alinhado
  - Deve seguir o alinhamento do maior tipo



Alinhamento de Memória

- Endereço inicial também tem que estar alinhado
  - Deve seguir o alinhamento do maior tipo



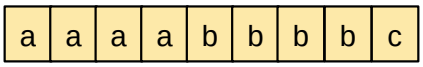
Obs: malloc() no Linux retorna sempre um ponteiro alinhado com 16, para que a memória possa ser usada para guardar qualquer objeto.

Alinhamento de Memória

- O tamanho de uma estrutura deve garantir o alinhamento para um array de estruturas

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

```
struct s s1;
```



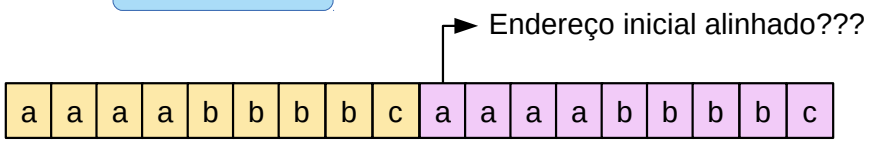
Obs: incorreto!

Alinhamento de Memória

- O tamanho de uma estrutura deve garantir o alinhamento para um array de estruturas

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

```
struct s s1[2];
```



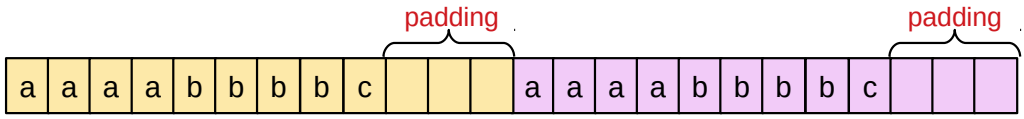
Obs: incorreto!

Alinhamento de Memória

- O tamanho de uma estrutura deve garantir o alinhamento para um array de estruturas

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

```
struct s s1[2];
```



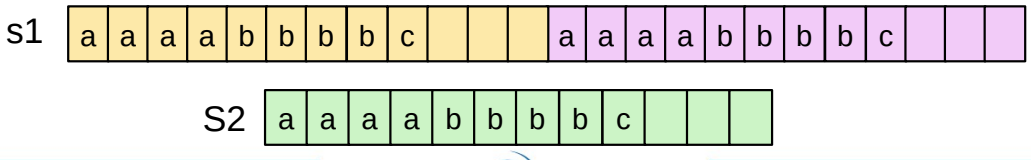
Alinhamento de Memória

- O tamanho de uma estrutu
  alinhamento para um arra

Padding no final é colocado em toda estrutura, não só em caso de array!!!

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

```
struct s s1[2];
struct s S2;
```



Alinhamento de Memória

- Temos que adicionar *padding* no fim da estrutura
- Tamanho final da estrutura deve ser múltiplo do maior tipo
- Exemplos:

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

Tamanho múltiplo de 4 (int)

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

Tamanho múltiplo de 8 (long)

```
struct s {
  char *a;
  short b;
  char  c;
};
```

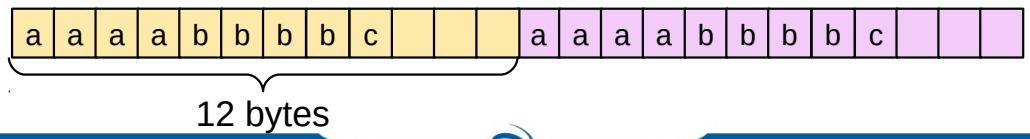
Tamanho múltiplo de 8 (ponteiro)

Alinhamento de Memória

- Temos que adicionar *padding* no fim da estrutura
- Tamanho final da estrutura deve ser múltiplo do maior tipo
- Exemplos:

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

Tamanho múltiplo de 4 (int)

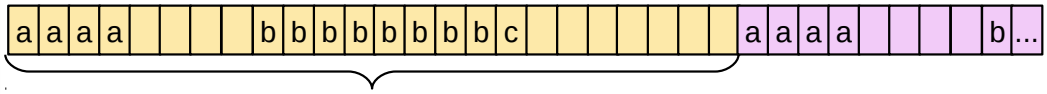


Alinhamento de Memória

- Temos que adicionar *padding* no fim da estrutura
- Tamanho final da estrutura deve ser múltiplo do maior tipo
- Exemplos:

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

Tamanho múltiplo de 8 (long)



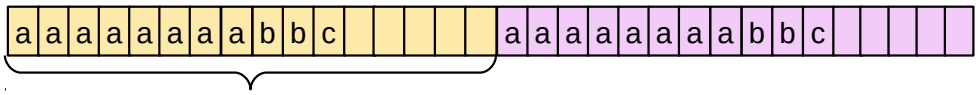
24 bytes

Alinhamento de Memória

- Temos que adicionar *padding* no fim da estrutura
- Tamanho final da estrutura deve ser múltiplo do maior tipo
- Exemplos:

```
struct s {
  char *a;
  short b;
  char  c;
};
```

Tamanho múltiplo de 8 (ponteiro)



16 bytes

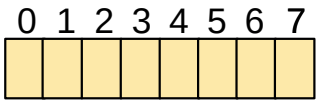
União

União

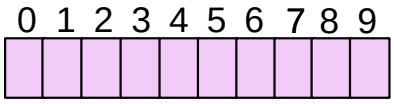
- União permite armazenar valores diferentes durante diferentes momentos do programa
- Os valores compartilham memória
  - Ou seja, um sobrescreve o outro
- O tamanho da união é do seu maior tipo

União

```
union u {
  int i;
  long l;
  char c;
};
```



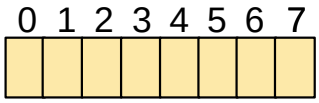

```
union u {
  int i;
  long l;
  char s[10];
};
```



União

```
union u {
  int i;
  long l;
  char c;
};
```

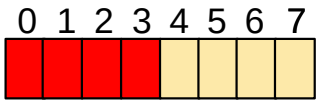
```
union u var;

var.i = 10;
var.c = 5;
var.l = 1024;
```



União

```
union u {
  int i;
  long l;
  char c;
};
```

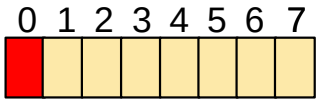
```
union u var;

var.i = 10;
var.c = 5;
var.l = 1024;
```



União

```
union u {
  int i;
  long l;
  char c;
};
```

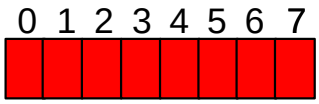
```
union u var;

var.i = 10;
var.c = 5;
var.l = 1024;
```



União

```
union u {
  int i;
  long l;
  char c;
};
```

```
union u var;

var.i = 10;
var.c = 5;
var.l = 1024;
```

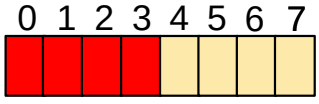


União

- C não controla o que é lido ou escrito
- É responsabilidade do programador

```
union u {
  int i;
  long l;
  char c;
};
```

```
union u var;
int x;

var.l = 1024;
x = var.i;
```



Vai ler parte de um *long*

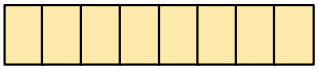
Exemplo

```
struct Circulo {
  int x;
  int y;
  double raio;
};
```

```
union Figura {
  struct Circulo c;
  struct Retangulo r;
};
```

```
struct Retangulo {
  int x1, y1;
  int x2, y2;
};
```

Tamanho?!?



Exemplo

```
struct Circulo {
  int x;
  int y;
  double raio;
};
```

```
union Figura {
  struct Circulo c;
  struct Retangulo r;
};
```

VS

```
struct Figura {
  struct Circulo c;
  struct Retangulo r;
};
```

```
struct Retangulo {
  int x1, y1;
  int x2, y2;
};
```