

# *Arrays* bidimensionais, estruturas e enumerados

Programação em Sistemas Computacionais

João Pedro Patriarca ([joao.patriarca@isel.pt](mailto:joao.patriarca@isel.pt)), Gabinete F.0.23 do edifício F

ISEL, ADEETC, LEIC

# Agenda

---

- *Arrays* bidimensionais
- Estruturas
- Enumerados

# Agenda

---

- *Arrays* bidimensionais
- Estruturas
- Enumerados

# Arrays bidimensionais

Bib: (A), cap. 5 {.7-.9}

- Mesmas características que um *array* unidimensional:
  - Sequência de elementos de um tipo em memória
- Reserva memória para N\*M elementos de mesmo tipo
- A dimensão N e M são constantes e conhecidas em tempo de compilação
- Não permite a afetação de *arrays*
- Permite a inicialização explícita

```
int a[3][5] = {  
    {1, 2, 3, 4, 5},  
    {9, 8, 7},  
};
```

Modelo lógico

a:	[0]				[4]	
[0]	1	2	3	4	5	} Linhas
[1]	9	8	7	0	0	
[2]	0	0	0	0	0	
} Colunas						

Modelo em memória

	a:	
a[0][0]	1	0x7fff3c1fc9c0
a[0][1]	2	0x7fff3c1fc9c4
	...	
a[0][4]	5	0x7fff3c1fc9d0
a[1][0]	9	0x7fff3c1fc9d4
	...	
a[2][4]	0	0x7fff3c1fc9f8

# Arrays bidimensionais - exemplo

```
void f0() {  
    int a[3][5] = { {1, 2, 3, 4, 5}, {9, 8, 7} };  
    printf("a = &a[0][0] : %p = %p\n", (void*)a, (void*)&a[0][0]);  
    printf("a[0] = &a[0][0] : %p = %p\n", (void*)a[0], (void*)&a[0][0]);  
    printf("a[1] = &a[1][0] : %p = %p\n", (void*)a[1], (void*)&a[1][0]);  
    printf("a[2] = &a[2][0] : %p = %p\n", (void*)a[2], (void*)&a[2][0]);  
    printf("dif(a[1], a[0]) = dif(a[2], a[1]) : %ld bytes = %ld bytes\n",  
        (a[1]-a[0])*sizeof(int), (a[2]-a[1])*sizeof(int));  
}
```




```
a = &a[0][0] : 0x7fff3c1fc9c0 = 0x7fff3c1fc9c0  
a[0] = &a[0][0] : 0x7fff3c1fc9c0 = 0x7fff3c1fc9c0  
a[1] = &a[1][0] : 0x7fff3c1fc9d4 = 0x7fff3c1fc9d4  
a[2] = &a[2][0] : 0x7fff3c1fc9e8 = 0x7fff3c1fc9e8  
dif(a[1], a[0]) = dif(a[2], a[1]) : 20 bytes = 20 bytes
```

# Array bidimensional como parâmetro de função - exemplo

```
#define LINES 3
#define COLUMNS 5
void f2() {
    int a[LINES][COLUMNS];
    printf("&a[0][0] = %p\n", (void*)a);
    printf("sizeof(a) = %ld bytes\n",
           sizeof(a));
    fill_array(a);
    print_array(a);
    printf("a[1][4] = %d\n", a[1][4]);
    a[2][-1] = 1;
    printf("a[1][4] = %d\n", a[1][4]);
}
```

```
void fill_array(int a[LINES][COLUMNS]) {
    int i, j;
    printf("sizeof(a) = %ld bytes\n", sizeof(a));
    printf("sizeof(a[0]) = %ld bytes\n", sizeof(a[0]));
    printf("&a[0][0] = %p\n", (void*)&a[0][0]);
    for (i = 0; i < LINES; i++)
        for (j = 0; j < COLUMNS; j++) a[i][j] = random();
}

void print_array(int a[][COLUMNS]) {
    int i, j;
    for (i = 0; i < LINES; i++)
        for (j = 0; j < COLUMNS; j++)
            printf("a[%d][%d] = %d\n", i, j, a[i][j]);
}
```



```
&a[0][0] = 0x7ffd545f1690
sizeof(a) = 60 bytes
sizeof(a[1]) = 20 bytes
sizeof(a) = 8 bytes
&a[0][0] = 0x7ffd545f1690
```

```
a[0][0] = 1804289383
...
a[2][4] = 2044897763
a[1][4] = 1189641421
a[1][4] = 1
```

# Array bidimensional vs *array* de ponteiros para *arrays*

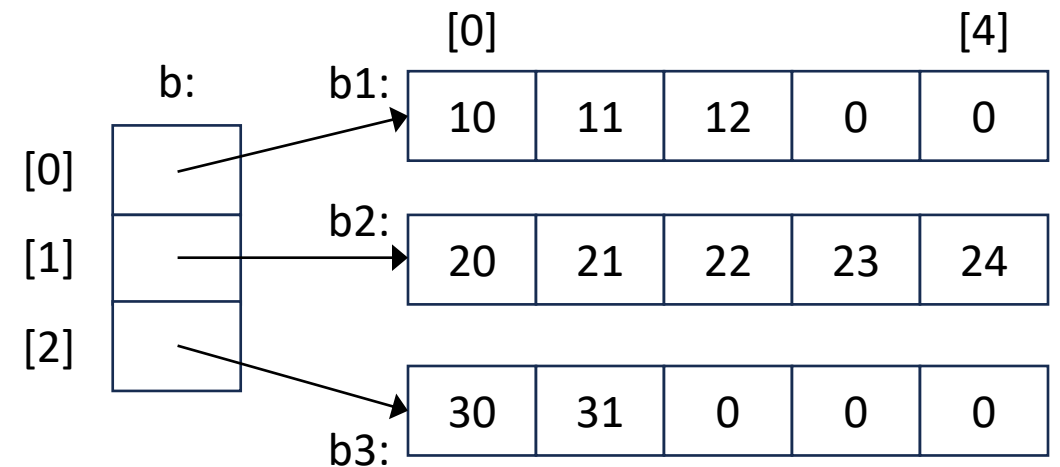
- Um *array* bidimensional é diferente de um *array* de ponteiros para *arrays*, do ponto de vista do modelo em memória

```
int b1[5] = {10, 11, 12};  
...  
int b2[5] = {20, 21, 22, 23, 24};  
...  
int b3[5] = {30, 31};  
int * b[3] = {b1, b2, b3};  
int v = b[1][3]; // v = 23
```

Modelo em memória


b1[0]	10	0x7fff3c1fc950
	...	
b2[0]	20	0x7fff3c1fc970
	...	
b3[0]	30	0x7fff3c1fc990
	...	
b[0]	0x7fff3c1fc950	0x7fff3c1fc9b0
b[1]	0x7fff3c1fc970	0x7fff3c1fc9b8
b[2]	0x7fff3c1fc990	0x7fff3c1fc9c0

Modelo lógico



# Arrays de ponteiros para *arrays* - exemplo

```
int b1[5] = {10, 11, 12}; int dummy_space1[10]; int b2[5] = {20, 21, 22, 23, 24};
int dummy_space2[20]; int b3[5] = {30, 31}; int *b[3] = {b1, b2, b3};
printf("b1 = %p; b2 = %p; b3 = %p\n", (void*)b1, (void*)b2, (void*)b3);
printf("b[0] = &b[0][0] != &b[0]: %p = %p != %p\n", (void*)b[0], (void*)&b[0][0], (void*)&b[0]);
printf("b[1] = &b[1][0] != &b[1]: %p = %p != %p\n", (void*)b[1], (void*)&b[1][0], (void*)&b[1]);
printf("b[2] = &b[2][0] != &b[2]: %p = %p != %p\n", (void*)b[2], (void*)&b[2][0], (void*)&b[2]);
printf("dif(b[1], b[0]) = dif(b[2], b[1]) : %ld bytes = %ld bytes\n", b[1]-b[0], b[2]-b[1]);
printf("b[0][0] = *b[0] : %d = %d\n", b[0][0], *b[0]);
printf("b[0][1] = *(b[0]+1) : %d = %d\n", b[0][1], *(b[0]+1));
```



```
b1 = 0x7fff3c1fc950; b2 = 0x7fff3c1fc970; b3 = 0x7fff3c1fc990
b[0] = &b[0][0] != &b[0]: 0x7fff3c1fc950 = 0x7fff3c1fc950 != 0x7fff3c1fc9b0
b[1] = &b[1][0] != &b[1]: 0x7fff3c1fc970 = 0x7fff3c1fc970 != 0x7fff3c1fc9b8
b[2] = &b[2][0] != &b[2]: 0x7fff3c1fc990 = 0x7fff3c1fc990 != 0x7fff3c1fc9c0
dif(b[1], b[0]) = dif(b[2], b[1]) : 8 bytes = 8 bytes
b[0][0] = *b[0] : 10 = 10
b[0][1] = *(b[0]+1) : 11 = 11
```



# Agenda

---

- *Arrays* bidimensionais
- Estruturas
- Enumerados

# Estruturas

Bib: (A), cap. 6 {.1-.4}

- Tipo composto
  - Agrega elementos de tipos diferentes
  - Pode agregar outras estruturas
- Palavra reservada *typedef*
  - Define nomes alternativos para tipos
  - Útil no âmbito de estruturas
- Ao invés dos *arrays*, é permitida a cópia de estruturas
- Operador . (ponto) e -> (setinha) para acesso aos campos

```
typedef unsigned char uchar;
```

```
struct ex1 { uchar fld1; long fld2; };  
typedef struct {  
    float fld3;  
    struct ex1 ex1;  
} Ex2, *PEX2;  
struct ex1 ex1;  
Ex2 ex2 = {2.4, {10, 20}};
```

```
Ex2 ex3 = ex2;  
printf("ex3={%f, {%d, %ld}}\n",  
       ex3.fld3, ex3.ex1.fld1,  
       ex3.ex1.fld2);
```

```
PEX2 pex2 = &ex2;  
printf("pex2={%f, {%d, %ld}}\n",  
       pex2->fld3, pex2->ex1.fld1,  
       pex2->ex1.fld2);
```

# Estrutura - dimensão

- O que justifica a diferença de valores?

```
struct ex1 {  
    uchar fld1;  
    long fld2;  
};
```

```
struct ex1 ex1;  
...  
printf("sizeof(ex1) = %ld bytes, sizeof(ex1.fld1) + sizeof(ex1.fld2) = %ld bytes\n",  
    sizeof(ex1), sizeof(ex1.fld1) + sizeof(ex1.fld2));
```



```
sizeof(ex1) = 16 bytes, sizeof(ex1.fld1) + sizeof(ex1.fld2) = 9 bytes
```

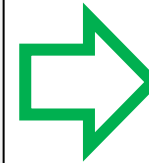
- Por omissão, o compilador privilegia a eficiência de execução em relação a eficiência do espaço ocupado em memória

# Estruturas – dimensões – outros exemplos

---

```
typedef struct { long fld1; uchar fld2; } Ex3;  
typedef struct { int fld1; uchar fld2; } Ex4;  
typedef struct { int fld1; long fld2; uchar fld3; } Ex5;  
typedef struct { long fld1; int fld2; uchar fld3; } Ex6;
```

```
printf("sizeof(Ex3) = %ld bytes\n", sizeof(Ex3));  
printf("sizeof(Ex4) = %ld bytes\n", sizeof(Ex4));  
printf("sizeof(Ex5) = %ld bytes\n", sizeof(Ex5));  
printf("sizeof(Ex6) = %ld bytes\n", sizeof(Ex6));
```



```
sizeof(Ex3) = 16 bytes  
sizeof(Ex4) = 8 bytes  
sizeof(Ex5) = 24 bytes  
sizeof(Ex6) = 16 bytes
```

- Por omissão, a dimensão de uma estrutura é múltipla da dimensão do maior campo
  - Garante o alinhamento adequado para todos os campos numa sequência de objetos estrutura
  - A ordem de definição dos campos torna-se relevante

# Estruturas como parâmetros e retorno de funções

- Uma estrutura pode ser passada como parâmetro a uma função, por valor ou por referência
- Uma estrutura pode ser retornada por uma função, por valor ou por referência

```
typedef struct {  
    int i; char c;  
} Test;
```

```
void print_struct(const Test *st,  
                 const char * msg) {  
    printf("%s {i = %d, c = '%c'}\n",  
          msg, st->i, st->c);  
}  
Test add_by_value(Test st1, Test st2) {  
    printf("&st1 = %p; &st2 = %p\n",  
          (void*)&st1, (void*)&st2);  
    st1.i += st2.i;  
    return st1;  
}  
Test * add_by_ref(Test *st1, Test *st2) {  
    printf("st1 = %p; st2 = %p\n",  
          (void*)st1, (void*)st2);  
    st1->i += st2->i;  
    return st1;  
}
```

```
void f4() {  
    Test t[] = {{100, 'a'}, {200, 'b'}}, t1;  
    printf("&t[0] = %p; &t[1] = %p\n",  
          (void*)&t[0], (void*)&t[1]);  
    t1 = add_by_value(t[0], t[1]);  
    print_struct(&t[0], "t[0] = ");  
    t[0] = t1;  
    print_struct(&t[0], "after t[0]=t1; t[0]=");  
    add_by_ref(&t[0], &t[1]);  
    print_struct(&t[0], "t[0] = ");  
}
```



```
&t[0] = 0x7ffffcb8b8250; &t[1] = 0x7ffffcb8b8258  
&st1 = 0x7ffffcb8b8228; &st2 = 0x7ffffcb8b8220  
t[0] = {i = 100, c = 'a'}  
after t[0]=t1; t[0]= {i = 300, c = 'a'}  
st1 = 0x7ffffcb8b8250; st2 = 0x7ffffcb8b8258  
t[0] = {i = 500, c = 'a'}
```

# Agenda

---

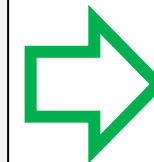
- *Arrays* bidimensionais
- Estruturas
- Enumerados

# Enumerados

Bib: (A), cap. 2.3

- Associa símbolos a valores inteiros
  - Preferível ao `#define` porque permite validação sintática por parte do compilador
- Por omissão, o primeiro símbolo do enumerado tem associado o valor 0
  - Os símbolos podem ser associados explicitamente a valores inteiros
- Por omissão, o símbolo subsequente é incrementado de um relativamente ao símbolo anterior
- Permite vários símbolos com o mesmo valor

```
typedef enum {  
    a, b, c='c', d, e, f, w='w', x, y, z='w'  
} SomeAlphaCodesEnum;  
void f5() {  
    SomeAlphaCodesEnum letter = a;  
    printf("a=%d\nc=%d\nd=%d\nw=%d\nx=%d\nz=%d\n",  
        letter, c, d, w, x, z);  
}
```



```
a=0  
c=99  
d=100  
w=119  
x=120  
z=119
```

# Enumerados – exemplo *days\_of\_month*

```
typedef enum months_t {
    Jan, Feb, Mar, Abr, Mai, Jun, Jul, Ago, Set, Out, Nov, Dez
} Months;

int is_leap_year(int year) {
    return (year % 4 == 0 && year % 100 != 0) || year % 400 == 0;
}

int days_of_month(Months m, int year) {
    const int months[] = {31, 28, 31, 30, 31, 30, 31,
                          31, 30, 31, 30, 31};
    return (m == Feb && is_leap_year(year)) ? 29 : months[m];
}

void f6_days_of_month(char * year_str) {
    const char * months_str[] = {"Jan", "Fev", "Mar", "Abr",
                                  "Mai", "Jun", "Jul", "Ago", "Set", "Out", "Nov", "Dez"};
    int year = atoi(year_str); /* não valida year_str */
    printf("Ano: %d\n", year);
    for (Months i = Jan; i <= Dez; i++)
        printf("\t%s: %d days\n", months_str[i], days_of_month(i, year));
}
```

Ano: 2023

Jan:	31 days
Fev:	28 days
Mar:	31 days
Abr:	30 days
Mai:	31 days
Jun:	30 days
Jul:	31 days
Ago:	31 days
Set:	30 days
Out:	31 days
Nov:	30 days
Dez:	31 days



# Exercício

Realize a função *words\_histogram* que escreve num ficheiro de texto o número de ocorrências de todas as palavras presentes, igualmente, num ficheiro de texto. Os ficheiros são especificados por via do redirecionamento do *standard input* e do *standard output*.

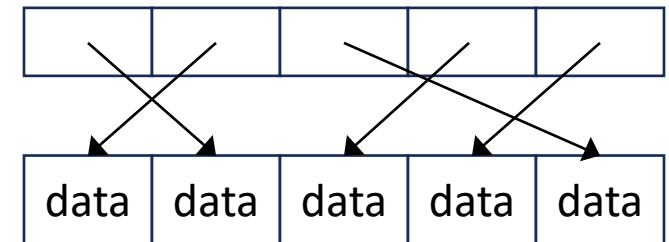
```
void words_histogram(FILE* fin, FILE *fout);
```

- a) Versão 1: *array* de estruturas
- b) Versão 2: *array* de ponteiros para estrutura
- c) Versão 3: lista simplesmente ligada
- d) Versão 4: árvore binária

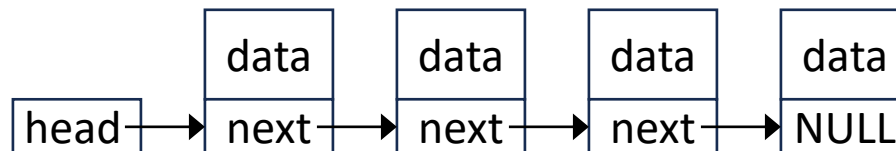
Versão1:



Versão2:



Versão3:



Versão4:

