

# Tipos com vírgula flutuante (*float* e *double*), *Bit fields* e *Unions*

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

---

- Codificação dos tipos *float* e *double* – norma IEEE754
- *Bit fields*
- Estrutura programática *union*

# Agenda

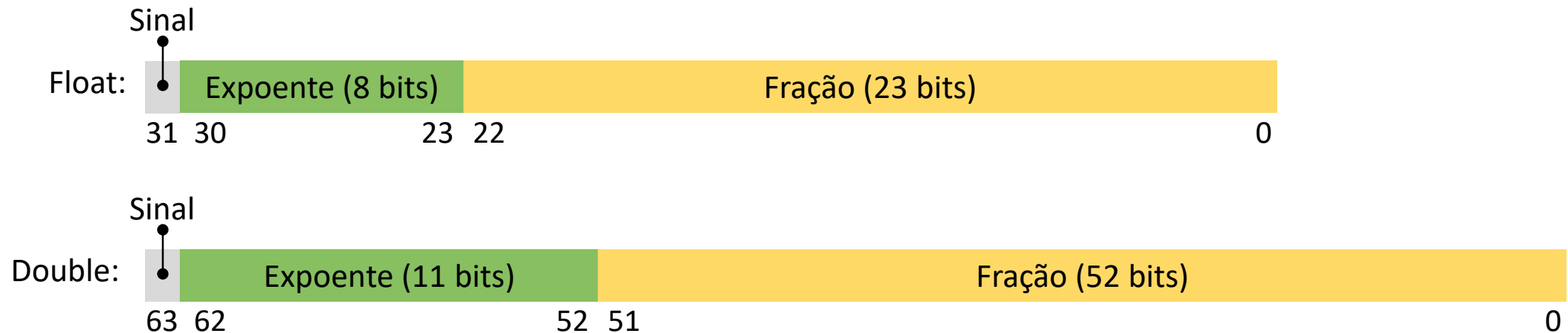
---

- Codificação dos tipos *float* e *double* – norma IEEE 754
- *Bit fields*
- Estrutura programática *union*

# Norma IEEE754

Bib: (A), cap. 2

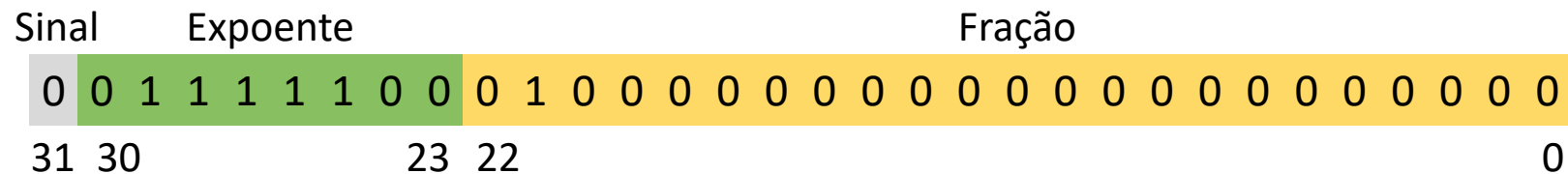
Tipo	Utilização	Dimensão (norma IEEE754)	Expoente (bits)	Mantissa (bits)	Intervalo de valores
<i>float</i>	Valores reais pequenos	32 bits (4 bytes)	8	23	$\pm 1.40 \times 10^{-45}..$ $\pm 3.40 \times 10^{+38}$
<i>double</i>	Valores reais grandes	64 bits (8 bytes)	11	52	$\pm 4.94 \times 10^{-324}..$ $\pm 1.76 \times 10^{+308}$



## Exemplo de codificação de um *float* (valor normalizado)

---

- $v = \pm m \times 2^e$
- $m = 1.M$  (valor normalizado:  $1 < m < 2$ ),  $e = E - 127$
- $M$  é a mantissa, valendo cada bit:  $2^{-1} + 2^{-2} + 2^{-3} + \dots + 2^{-22} + 2^{-23}$
- $E$  é o expoente, valendo cada bit:  $2^7 + 2^6 + 2^5 + \dots + 2^1 + 2^0$
- Exemplo:



- $M = 2^{-2} = 0.25 \Rightarrow m = 1.25$
- $E = 2^6 + 2^5 + 2^4 + 2^3 + 2^2 = 124 \Rightarrow e = 124 - 127 = -3$
- $v = +1.25 \times 2^{-3} = 0.15625$

# Representação de valores especiais

- Zero

- Existe a noção de  $+0$  e de  $-0$ , dependendo do valor do sinal

S 0

- Infinito

- Existe a noção de  $+\infty$  e de  $-\infty$ , dependendo do valor do sinal

S 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

- Indeterminado (NaN – *Not a Number*)

- Irrelevante o valor do sinal

- 1 1 1 1 1 1 1 1 Diferente de 0

# Operações com valores especiais

---

Operação	Resultado
$n \div \pm Infinity$	0
$\pm Infinity \times \pm Infinity$	$\pm Infinity$
$\pm nonZero \div \pm 0$	$\pm Infinity$
$\pm finite \times \pm Infinity$	$\pm Infinity$
$Infinity + Infinity$ $Infinity - -Infinity$	$+Infinity$
$-Infinity - Infinity$ $-Infinity + -Infinity$	$-Infinity$
$\pm 0 \div \pm 0$	NaN
$\pm Infinity \div \pm Infinity$	NaN
$\pm Infinity \times 0$	NaN
NaN == NaN	False

# Exercícios

---

- Regra: as implementações internas apenas podem usar operações aritméticas e lógicas sobre inteiros. Qualquer operação de vírgula flutuante invalida a resolução do exercício

- Escreva a função *float\_mul\_by\_2*, para retornar a multiplicação de valor do tipo *float* por 2

```
int float_mul_by_2(float fv);
```

- Escreva a função *float\_int\_cmp*, para realizar a comparação da componente inteira de um valor do tipo *float* com um valor do tipo *int*. A função retorna +1 se *fv* maior que *iv*, -1 se *fv* menor que *iv* e 0 se *fv* igual a *iv*

```
int float_int_cmp(float fv, int iv);
```

- Escreva a função *float\_round\_to\_int*, para arredondar um valor do tipo *float* para o inteiro mais próximo

```
int float_round_to_int(float fv);
```



# Agenda

---

- Codificação dos tipos *float* e *double* – norma IEEE 754
- *Bit fields*
- Estrutura programática *union*

# Bit fields

---

- Capacidade de definir e aceder a campos dentro de uma *palavra*
- Uma *palavra* é uma unidade de armazenamento, representada por uma sequência de bits
- Os campos são definidos no contexto de uma estrutura com a indicação da largura de cada campo em bits
  - Os campos são sempre definidos como inteiros (*signed/unsigned*)
  - `<integer type> <field_name>: <bits_width>;`
- Permite a definição de campos sem nome apenas para introduzir espaço (*gap*) entre campos com nome
  - A introdução de espaço pode servir para que um campo não atravessasse uma unidade de armazenamento
- Não permite obter o endereço de um campo

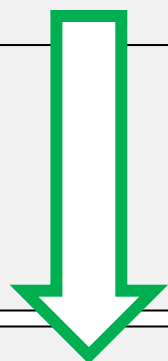
# Exemplo com *bit-fields*

```
#include "common.h"

typedef struct {
    int    a: 12;
    int    : 4;
    uint   b: 8;
    int    c: 2;
    int    d: 4;
    int    e: 32;
} BFEEx;

void out_field(const char * msg, BFEEx * pv) {
    printf("%s => a=%d, b=%d, c=%d, d=%d, e=%d\n",
        msg, pv->a, pv->b, pv->c, pv->d, pv->e);
}

int main() {
    BFEEx v1 = {1024, 255, -1, -2, 3};
    out_field("After creation", &v1);
    v1.a = v1.a + v1.b; out_field("a=a+b", &v1);
    v1.b = v1.b + v1.e; out_field("b=b+e", &v1);
    v1.c = v1.c - v1.b; out_field("c=c-b", &v1);
    v1.e = v1.a * v1.d; out_field("e=a*d", &v1);
    return 0;
}
```



After creation => a=1024, b=255, c=-1, d=-2, e=3  
a=a+b => a=1279, b=255, c=-1, d=-2, e=3  
b=b+e => a=1279, b=2, c=-1, d=-2, e=3  
c=c-b => a=1279, b=2, c=1, d=-2, e=3  
e=a\*d => a=1279, b=2, c=1, d=-2, e=-2558

# Agenda

---

- Codificação dos tipos *float* e *double* – norma IEEE 754
- *Bit fields*
- Estrutura programática *union*

# Estrutura programática *union*

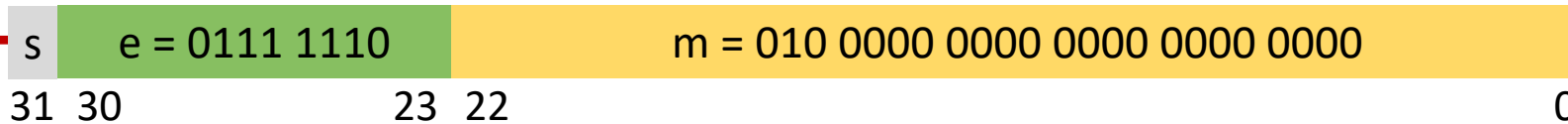
- Semelhante em tudo com uma estrutura (*struct*) mas com a diferença de que todos os campos se sobrepõem
- A dimensão de uma *union* corresponde à dimensão do maior campo
- Permite a manipulação de diferentes tipos de dados para o mesmo valor em memória

```
typedef union {  
    char a; short b; int c; long d; float e; double f;  
} UnionEx;  
  
int main() {  
    UnionEx v = {0};  
    printf("sizeof(UnionEx)=%ld\n", sizeof(UnionEx));  
    printf("&v=%p\n&v.a=%p\n&v.b=%p\n&v.e=%p\n",  
        (void*)&v, (void*)&v.a, (void*)&v.b, (void*)&v.e);  
    v.c = 0x12345678;  
    printf(" v.c=0x%lx\n v.a=0x%x\n", v.c, v.a);  
    return 0;  
}
```

```
sizeof(UnionEx)=8  
&v=0x7fff33ec6350  
&v.a=0x7fff33ec6350  
&v.b=0x7fff33ec6350  
&v.e=0x7fff33ec6350  
v.c=0x12345678  
v.a=0x78
```



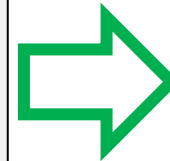
# Exemplo para extrair as componentes inteiras de um float



```
typedef union {
    struct {
        unsigned int m: 23;
        unsigned int e: 8;
        unsigned int s: 1;
    };
    float f;
} Float_BF;

int main() {
    Float_BF bf;
    bf.f = 0.15625;
    printf("s=%d | e=0x%02x | m=0x%06x\n",
        bf.s, bf.e, bf.m);
    // Individualiza cada componente do float
    int sign = bf.s, exp = bf.e - 127;
    int normalized_value = (1 << 23) | bf.m;
    printf("exp=%d, normalized_value=0x%06x\n",
        exp, normalized_value);

    return 0;
}
```



s=0 | e=0x7c | m=0x200000

exp=-3,  
normalized\_value=0xa00000