# 11ª Aula - Estruturas. 'typedef'. Bit Field. União.

# Programação

## Mestrado em Engenharia Física Tecnológica

Samuel M. Eleutério
sme@tecnico.ulisboa.pt

Departamento de Física
Instituto Superior Técnico
Universidade de Lisboa

# Estruturas

- Uma das entidades mais importantes, em **C**, é a **estrutura**.
- Uma **estrutura** permite **agrupar** num objecto um **conjunto de elementos** constituídos pelos mais variados tipos.
- Pode pensar-se numa **estrutura** como uma **ficha de informação**. Ela pode ter o nome, a idade, o telefone, a marca do carro, a altura, o peso, a cor dos olhos, as nossas habilitações, etc..
- Na sua **definição** cria-se um **tipo** (isto é, o nome pela qual será designada), a sua **constituição**, isto é, a descrição das variaveis nela contidas (com os respectivos tipos) e, eventualmente, declaram-se **variáveis** com esse tipo:

**struct designação** {
  **tipo**$_1$ **var**$_{11}$, **var**$_{12}$, ... ;
  .....
  **tipo**$_M$ **var**$_{M1}$, **var**$_{M2}$, ... ;
} **obj**$_1$, ... , **obj**$_N$ ;

# Estruturas - Exemplo

- A título de exemplo, podemos definir uma **estrutura**, que designamos por **particula**, que contenha algumas informações referentes às partículas físicas:

```
struct particula {
    char nome[80] ;
    int carga ;
    int paridade ;
    float spin ;
    double massa, vida_media ;
} ;
```

- E vamos, em seguida, **fazer a declaração** de duas variáveis:

```
struct particula part1, part2;
```

- Se quisermos declarar uma variável com os dados do **electrão**:

```
struct particula elect = {"electrao", -1, 1, 0.5, 0.511, 1.E40};
```

# Estruturas ('Prog14_01.c')

- Acabámos de criar uma **estrutura**, de declarar uma **variável** e de a **inicializar**. Mas como se pode **aceder** às suas **componentes**?
- Se quisermos saber a massa do electrão, temos antes de mais de indicar a **variável** declarada pela estrutura '**elect**' e dentro dela queremos a variável '**massa**', então a massa do electrão será:

  **elect.massa**

  Note-se que a variável '**elect.massa**' é um **double** e comporta-se exactamente como qualquer outro **double** que vimos até agora.
- Se estivermos a lidar com um ponteiro para uma estrutura em vez da própria estrutura, temos de ter em conta alguns pormenores.
- Ao passar o **ponteiro** da estrutura para uma **função** escrevemos:

  **func (&elect, ...)**
- E ao declararmos a função:

  **tipo func (struct particula *p1, ...)**

# Estruturas ('Prog14_02.c')

- Uma vez dentro da função, **p1** é um **ponteiro** para a estrutura.
- Se quisermos a **própria estrutura**, pedimos o **valor do ponteiro**, ou seja, ***p1**.
- De um modo idêntico ao que fizemos anteriormente, podemos referirmo-nos à **massa da partícula** como '(***p1**).**massa**'.
- Uma vez que as estruturas são objectos extremamente frequentes em **C**, era desejável que existisse uma notação, equivalente a esta mas, um pouco mais simples. Assim podemos escrever:

  $$(*p1).massa \iff p1->massa$$

  em que o símbolo '$->$' indica que nos estamos a referir ao **ponteiro** para uma estrutura e **não à própria estrutura**.
- Não esquecer que estamos a lidar com notações equivalentes e que a simplificação de escrita determina o seu uso.
- Como se verá, na esmagadora maioria dos casos da vida real, lidamos com **ponteiros para estrutura** e **não** com **estruturas**.

# Estruturas ('Prog14_03.c' a 'Prog14_05.c')

- Por vezes é conveniente, no **interior de uma estrutura**, ter um ponteiro para uma estrutura idêntica. Isto é:

```
struct particula {
    struct particula *anti ;
    char nome[80] ;
    .....}
```

- Apesar da estrutura **não estar** completamente definida, podemos fazer esta declaração, pois, ela apenas pressupõe um **ponteiro**. Isto é possível porque o **C** permite definir **tipos incompletos**.
- As **estruturas** também podem ser organizadas em **vectores**. Usando a estrutura '**particula**' podemos querer guardar a informação de um **conjunto de partículas**:

```
struct particula fermiao[50], bosao[50] ;
```

# Redefinir Tipos: 'typedef' ('Prog14_06.c')

- Para facilitar a **atribuição de tipos**, em **C**, existe um modo simples ('**typedef**') de lhes associar sinónimos.
- O comando '**typedef**' tem por forma:

  **typedef tipo nome_alternativo** ;

- Se desejarmos chamar **byte** a um **char** podemos escrever:

  **typedef char byte** ;
  **byte** v1, v2 ;

- De um modo idêntico, podemos designar por **elementar** a estrutura **struct particula** atrás definida:

  **typedef struct particula elementar** ;

- Apesar de '**typedef**' não criar novos tipos, encontra-se **muito frequentemente associado** à **criação de tipos**.
- É bastante frequente, no momento em que se cria uma estrutura, **associar-lhe** um **nome alternativo**.

# Redefinir Tipos: 'typedef' ('Prog17_01.c')

- Um exemplo da utilização de '**typedef**' associa a uma estrutura de dois reais o nome alternativo de '**complexo**':

```
typedef struct
{
  double real ;
  double imag ;
} complexo ;
```

- Outro exemplo, é uma estrutura em que se associa uma string com o seu comprimento. Vamos designá-la por '**String**':

```
typedef struct
{
  int len ;
  char str[256] ;
} String ;
```

# Bit Field (1) ('Prog16_01.c')

- Um '**bit field**' (**campo de bits**) é uma opção na definição de uma estrutura.
- Nesta opção podemos indicar ao compilador qual o **número de bits** que deve ser **reservado** para cada variável.

```
struct designação {
    tipo1 var1 : nbits1 ;
    .....
    tipoN varN : nbitsN ;
} ;
```

- No programa '**Prog16_01.c**' apresenta-se uma estrutura que representa uma figura geométrica que pode estar visível ou não no ecran (**pintado**) e para a qual de reserva um espaço para a **diagonal** e outro para a **cor**.

# Bit Field (2) ('Prog16_02.c')

- No programa '**Prog16_02.c**' mostra como se pode **reduzir o espaço ocupado** desde que se use apenas o **número de bits** necessário para cada informação.
- Considere-se então uma estrutura em que se pretende guardar como inteiros, a **data de nascimento** e o **sexo** de uma pessoa.
- O espaço exigido por cada uma dessas indicações é:

  **Dia: [1,31]**, **Mês: [1,12]**, **Ano: [-8192,8191]**, **Sexo: [0,1]**

- Então a estrutura pode ser definida como:

```
struct DataNascSexo {
  unsigned int  dia  : 5 ;      // 2^5  = 32
  unsigned int  mes  : 4 ;      // 2^4  = 16
  int           ano  :14 ;      // 2^14 = 16384
  unsigned int  sexo : 1 ;      // 2^1  = 2
} ;
```

# União ('Prog15_01.c')

- Existe em **C** um objecto capaz de utilizar o mesmo espaço de memória com diferentes tipos, a **união** (**union**).
- Note-se que esta **ocupação simultânea**, significa que **não se pode usar** simultaneamente os diferentes tipo pois eles são escritos no **mesmo sítio da memória**.
- O tipo de declaração é análogo ao de uma estrutura.
- O programa '**Prog15_01.c**' mostra a utilização de uma **união**, com um **int** e um **float**, ambos ocupando **4 bytes**:

```
union numeros {
    int i ;
    float x ;
}
```

# Posição de um Elemento na Estrutura ('14_07.c')

- Dada uma estrutura, é possível saber qual a **posição** em que cada um dos seus **membros** se encontra, isto é, **a quantos bytes** está do seu início.
- Esta tarefa é implementada pela seguinte **macro**:

  **size_t offsetof (type, member)**

- No programa '**Prog14_07.c**' utiliza-se '**offsetof**' para obter a posição da '**massa**' em '**elementar**' (**struct particula**).
- Nele é igualmente mostrado um processo que permite, **conhecida a posição de um membro da estrutura**, **alterar o seu valor** apontando um ponteiro directamente para ele.
- Para tal, acrescenta-se à **posição inicial** da variável o **número de bytes** a que esse membro se encontra do **início da estrutura**.
- Isto é feito recorrendo-se a um artifício em que se faz um '**casting**' a '**char**' e depois outro '**casting**' ao **tipo** a ele associado (no exemplo da **massa** será a um '**double**').