

# 15ª Aula - Funções.


## Programação

### Mestrado em Engenharia Física Tecnológica

Samuel M. Eleutério  
sme@tecnico.ulisboa.pt

Departamento de Física  
Instituto Superior Técnico  
Universidade de Lisboa

# Funções

- **Funções** são **porções de código** encapsuladas que admitem **variáveis como argumentos** e podem retornar um **valor**.
- As **funções** podem aparecer num programa como **pré-definidas** (e frequentemente organizadas em **bibliotecas**) ou **nele construídas** para desempenharem uma determinada tarefa.
- As **funções** são as pedras sobre as quais se **constroem** os **programas**. Uma cuidadosa planificação permitirá um **código mais flexível** e mais bem adaptado aos objectivos que se pretendem atingir.
- **Todas** as funções em **C** têm um **tipo** de retorno.
- A **definição** de uma função ('**func**') tem a seguinte sintaxe:  
$$\text{tipo}_f \text{ func} (\text{tipo}_1 \ x_1, \text{tipo}_2 \ x_2, \dots, \text{tipo}_n \ x_n) \{ \text{Corpo} \}$$
- A sua **declaração** (por exemplo, nos ficheiros '**.h**') é idêntica à sua definição mas não inclui o corpo e termina em '**;**'.
- Se o tipo da função ou dos seus argumentos não for declarado, é admitido como '**int**'. É **conveniente**, no entanto, fazê-lo. 

# Funções

- Como se disse, uma **função** tem sempre um **tipo de retorno** (mesmo que seja um do tipo '**void**').
- O tipo '**void**' aparece igualmente como **argumento** de uma **função** sem argumentos (**int func (void) {...}**).
- O retorno de uma **função** é feito com a instrução com o seu tipo '**return instrução**;' ou com '**return**;' quando é do tipo '**void**' (Neste caso, não é obrigatório incluí-lo). Note-se, no entanto, que pode haver mais do um '**return**' dentro de uma função.
- Quando uma **função** é executada os valores com que ela é chamada são **copiados** para os seus argumentos.
- Esses valores **não sofrem qualquer alteração no exterior** da função, uma vez que ela **trabalha** apenas sobre **cópias**.
- Em **C**, as **funções** são todas definidas ao mesmo **nível**, isto é, uma **função** não deve conter a **definição** de outra **função**.

# Funções - Passagem de Variáveis ('Prog27\_01.c')

- Há dois modos de **receber** informação da execução de **funções**:
  - 1 Directamente através do seu **return** ou das **variáveis globais**;
  - 2 Indirectamente através dos **argumentos** da função.
- **Directamente**:
  - Através de **variáveis globais**;
  - Através da **variável** associada ao **return** da **função**
- **Indirectamente**, através dos **ponteiros**, transferidos para a **função**, podemos alterar os **valores das variáveis**, uma vez que estamos a escrever nas **posições de memória** em que **esses valores se encontram**.
- Num ficheiro, podemos ter **variáveis globais**, desde que as **declaremos fora das funções**.

## Ponteiros para Funções ('Prog28\_01 e 02.c')

- Uma vez que cada **função** tem o seu próprio **endereço de memória**, é então possível ter **ponteiros** para **funções**.
- Assim, para **declararmos** um **ponteiro** para uma **função** que retorna um **'double'** e que tem o primeiro argumento **'int'** e o segundo **'double'**, escrevemos:

```
double (*f) (int x, double y);
```

Os parêntesis em **'(\*f)'** são essenciais, senão tínhamos uma **função** que retornava um ponteiro para um **'double'**.

- Nos programas **'Prog28\_01 e 02.c'** pode ver-se como se pode decidir apontar para uma **função** e usar o **ponteiro** como se fosse a **própria função** (directamente ou como argumento).
- Por idênticas razões, podemos ter, como **argumentos** de uma **função**, **ponteiros** para **funções**.

# Ponteiros para Funções

## ('Prog28\_03.c' e 'Prog28\_04\_Dir')

- Um aspecto interessante resultante da existência de **ponteiros** para **funções** é a sua integração em **estruturas**.
- Em '**Prog28\_03.c**' são apresentados diferentes aspectos da **utilização** de **estruturas**:
  - Podemos ver como se pode apontar para um **conjunto de funções** usando um **vector** de **ponteiros**;
  - Podemos ver que através da estrutura não só se podem executar operações com **guardar resultados** para posteriores utilizações;
  - A utilização de um '**enum**' para guardar a informação da operação escolhida e com ela actuar sobre outras variáveis.

## Ponteiros para Funções ( 'Prog28\_03.c' e 'Prog28\_04\_Dir' )

- Note-se ainda a utilização de uma **função** para **criar** uma estrutura e **inicializá-la** devidamente (**função criadora**).
- Apesar de neste caso não ser necessário, é bastante conveniente escrever-se igualmente uma **função** capaz de **apagar** a estrutura criada (**função destruidora**).
- A utilização correcta deste par **criação-destruição** é essencial ao **bom funcionamento** de um programa.
- Pode ainda ver-se como se usou a função '**getchar**' para garantir que o '**buffer**' do terminal fica **limpo** para uma nova leitura.
- Este simples exemplo mostra o mecanismo básico de implementação da programação por objectos.

# Funções com Número Variável de Argumentos

## ('Prog29\_01e2.c')

- Por vezes, há necessidade de escrever **funções** com um **número variável** de argumentos (por exemplo, a função '**printf**').
- A definição deste tipo de **funções** requer **macros** que se encontram definidos em '**stdarg.h**'.
- A sua declaração requer vários passos:
  - 1 Declarar o **protótipo** da **função** com os argumentos explícitos e **reticências** nos implícitos;
  - 2 Inicializar '**vlist**' com a função '**va\_start**': **va\_start** (**vlist**, **darg**). Em que '**darg**' representa o argumento antes das reticências.
  - 3 O processamento da lista faz-se usando a função '**va\_arg**' na forma **va\_arg** (**vlist**, **tipo**), em que '**tipo**' designa o tipo da variável a extrair;
  - 4 Quando toda a lista '**vlist**' foi processada tem de se usar a função '**va\_end**' na forma **va\_end** (**vlist**).