13ª Aula - Ponteiros. Enumerados. Variáveis e Constantes (I).

Programação
Mestrado em Engenharia Física Tecnológica

Samuel M. Eleutério
sme@tecnico.ulisboa.pt

Departamento de Física
Instituto Superior Técnico
Universidade de Lisboa

Ponteiros ('Prog18_01.c')

- Em **C** todas as variáveis podem ser encaradas de dois pontos de vista diferentes:
  - Pelo seu **endereço**, isto é, pela sua localização na memória;
  - Pelo seu **valor**.
- As variáveis que guardam a **localização na memória** do espaço atribuído às variáveis (**endereço**) chamam-se **ponteiros**.
- Quando declaramos um **ponteiro**, declaramos também o tipo de variável para o qual estamos a apontar (**tipo *nome;**).
  Exemplos:

  double a, *p, *x, y, *q, z ;
- Se há **variáveis ponteiros**, então também deverão existir **ponteiros** que **apontam** para **ponteiros**!:

  double **r ;

  e assim sucessivamente... **double ***s**, etc..

Ponteiros (II)

- Ao utilizar **variáveis dimensionadas** (Ex: **char cnome**[80];), a variável assim declarada ('**cnome**') é um ponteiro para o início da zona de memória que lhe está reservada (neste caso, 80 bytes).
- As **variáveis dimensionadas podem ser inicializadas** no momento em que são definidas.
- Quando um **vector é inicializado** o número de elementos dele é o número de elementos indicado. Então não há **necessidade** de o explicitar:

  double t[ ] = {1., 2., 3., 4., 5., 6.};

  mas não é válido escrever simplesmente '**double t**[ ];'.
- Ao fazermos a **declaração** para objectos de **maiores dimensões** (matrizes, etc.), podemos escrever:

  int m[ ][3] = {{1,2,3},{4,5,6},{7,8,9},{10,11,12}};

  em que o último '[3]' indica que os valores, na matriz, deverão estar **agrupados** em linhas de **3 colunas**.

Ponteiros (III)

- Acontece, por vezes, que precisamos de **apontar** para objectos de que **não sabemos** ainda o tipo.
- Para isso foi criado o tipo '**void ***'.
- Um ponteiro para '**void**' pode assumir o **valor** de um ponteiro de qualquer outro tipo, e, inversamente, pode ser atribuído a **qualquer outro ponteiro**.
- Resta agora saber que operações são permitidas sobre ponteiros. Vejamos quais os **operadores** específicos que sobre eles actuam:
  - **Operador de referência** ('**&**'): quando aplicado a uma **variável** fornece o seu **endereço** na memória;
  - **Operador de derreferência** ('*'): quando aplicado a um **ponteiro** retorna o **valor** guardado na sua zona de memória;
  - **Operador de derreferência para estruturas** ('->'): retorna o valor do elemento da estrutura indicado;

Ponteiros (IV)

- Para além dos operadores atrás referidos são possíveis as **operações** de **soma** e **subtração**.
- Se somarmos a um **ponteiro** '**1**', isso significa que avançamos de um número de **bytes** correspondentes ao espaço ocupado por **um elemento** do tipo considerado:

  $*(p+n) \equiv p[n]$

  ou, inversamente,

  $\&p[n] \equiv p+n$

- Se, por exemplo, quisermos **ler** o valor de **v[2]**, usando a função '**scanf**', podemos escrever:

  scanf ("%f", &(**v[2]**)); $\iff$ **scanf**("%f", **v+2**);

- São válidas as operações com '++', '−−', '+=', '−='.
- O **resultado** das operações entre ponteiros depende da máquina e pode ser um **int** ou um **long int**.

Conjuntos Enumerados de Inteiros ('Prog13_01.c')

- Quando que deseja utilizar listas de valores inteiros aos quais se deseja associa um **nome**, é cómoda a utilização de **enumerados**.
- Os **enumerados** declaram-se listando nomes separados por vírgulas. Se não se derem mais indicações, o primeiro toma o valor '**0**' e os valores crescem de '**1**' em '**1**'.
- No entanto, podemos dar **designações diferentes** ao **mesmo valor**. Nesses casos é necessário **especificá-lo** explicitamente.
- Note-se que ao atribuirmos **explicitamente um valor**, o valor seguinte será o seu valor mais '**1**', mesmo que a ordem inicial seja com isso alterada.
- Os **enumerados** podem ser usados conjuntamente com os **inteiros**. Embora alguns compiladores não o exijam, é conveniente usar **sempre** o **molde** (**casting**) apropriado.

Tipos Básicos

Há dois grandes tipos de variáveis numéricas: tipo **inteiro** e tipo **real**.

Variáveis de tipo inteiro:

- **char:** ocupa **1 byte** (8 bits $= 2^8 = 256$) de memória. Pode ser usado para guardar um caracter ou um inteiro de 1 byte;
- **int:** ocupa **4 bytes** ($4 \times 8 = 32$ bits $= 2^{32} = 4\,294\,967\,296$) de memória. (Em certos casos pode ainda corresponder a **2 bytes**). Pode igualmente ser usado para guardar caracteres (formatos de caracteres Unicode – ver '**http://www.unicode.org/**').
- O tipo inteiro pode aparecer como **short int** (**2 bytes**), **long int** (**4 bytes**) ou **long long int** (**8 bytes**).
- Qualquer destes tipos podem ser definidos **com ou sem sinal**. Neste último caso a declaração é precedida por '**unsigned**'.
- Na representação binária de inteiros, os **valores negativos** são assinalados com **o primeiro bit à esquerda** marcado a '**1**'.

Tipos Básicos

Variáveis de tipo real:

- **float:** ocupa 4 bytes (precisão simples);
- **double:** ocupa 8 bytes (precisão dupla).

A representação de um **real** e feita de acordo com a expressão:

$$x = (-1)^s \times 2^{(E-B)} \times m$$

em que '**s**' é o sinal, '**m**' a mantissa e '**B**' o bias do expoente (E $\geq$ 0).

- Para um real em **precisão simples** (**float**) tem-se que '**s**' é o **primeiro bit** à esquerda, o **expoente** ('**E**') é representado pelos **8** bits seguintes, com **B** (**bias**) igual **127** e os restantes (**23** bits) são a **mantissa**, '**m**' (norma IEEE 754):

  0 01011101 10101000101110001010011
  s E m

- Para um real em **dupla precisão** (**double**), o sinal é igualmente o **primeiro bit**, o **expoente** usa **11** bits, o **bias** é **1023** e a **mantissa 52** bits.

Representações de Texto

- Anteriormente a cada **letra** fizemos corresponder um '**char**'.
- No entanto, esse tipo de notação é **insuficiente** para a representação de **outros** tipos de caracteres.
- O **chinês**, o **japonês**, o **coreano**, o **egípcio antigo**, etc., são representados por mais caracteres do que aqueles que podemos registar num byte.
- **Duas** estratégias são possíveis para resolver esta questão:
  - **multibyte (vários bytes)**: Fazemos a representação usando mais do que um '**char**', **sempre que necessário**
  - **wide char (caracteres largos)**: Usamos tipos com mais do que 1 byte (2 ou 4 bytes).
- Em '**locale.h**', encontram-se as definições associadas ao idioma, opções locais (moeda, estrutura de datas, etc.). Esse tipo de informações aparecem na literatura com a designação de '**internacionalização**' ou '**locales**'.

Representações de Texto

- As soluções de tipo '**wide char**', têm a vantagem de constituirem um **formato único**.
- No entanto, programas escritos para **1 byte por char** podem **não funcionar** correctamente neste ambiente.
- Por outro lado, um ambiente '**wide char**' de **2** ou **4** bytes, traduzir-se-ia, em termos de internet, em **acréscimos** muito significativos de **tráfego**.
- Numa codificação '**multibyte**', só se usa mais do que 1 byte quando tal é necessário.
  Para tal, existem **codigicações específica** que forçam à leitura de mais de 1 byte (**shift sequencies** – sequências de alteração) e que fixam o conjunto de bytes que se segue (**shift state** – estado alterado).
- Um exemplo desta codificação é o '**UTF-8**'.
- As **normas** de caracteres são presentemente desenvolvidas pelo '**Unicode Consortium**' (ver 'http://www.unicode.org/').

Variáveis Constantes – Inteiros ('Prog19_01.c')

- Qualquer quantidade que **não varia** durante a execução de um programa é uma **constante**.
- Apesar de o **valor** de um número ser o **mesmo**, ele pode ser representado de **diferentes maneiras**.
- Existem três representações de inteiros: decimal, hexadecinal e octal. A sua representação é constituída por '*prefixo-valor-sufixo*'.
  - **Decimais:** não podem iniciar-se por '**0**', devido à notação octal.
  - **Octal:** inicia-se obrigatoriamente por um '**0**'. Os algarismo válidos são '**0**' a '**7**'.
  - **Hexadecimais:** iniciam obrigatoriamente por '**0x**', são caracteres válidos '**0123456789ABCDEF**'. As minúsculas '**abcdef**' também são válidas.

  Podem ter como sufixos '**l**' ou '**L**' para '**long int**' e '**u**' ou '**U**' para '**unsigned int**'.

Variáveis Constantes – Reais e Literais

- Constituição de um real: **parte_int.parte_frac-exp-suf**. Para se considerar um real é necessário indicar o expoente ou o ponto '**.**'.
- Para '**double**' não há qualquer sufixo; para '**float**' o sufixo é '**f**' (ou '**F**'); para '**long double**' usa-se o sufixo '**l**' (ou '**L**').

Existem dois modos para literais:

- De **um só caracter**. Exemplos: '**\n**' (nova linha), '**%%**' (percentagem), '\\' (traço para trás), '**\"**' (aspas), etc..
- De **strings** (cadeias de caracteres). Exemplos:
  - **char st[6]** = {'S','o','d','i','o','\0'} ;
  - **char st[6]** = "Sodio" ;
  - **char *st** = "Sodio" ;
  - **char *grupo[40]** = {"Alcalino", "Terroso"} ;
  - **char *grupo[]** = {"Alcalino", "Terroso"} ;

Constantes Especiais ('Prog20_01.c')

- Em **C** existem algumas constantes especialmente úteis. Elas encontram-se definidas nas files '**.h**'.
- **EOF** (**End Of File**): indica que se chegou ao **fim de uma file**. Pode ser retornado pela função '**fscanf**' e ser usado num ciclo, por exemplo, para terminar uma leitura:

  while (**fscanf** (**fich**, "**%f**", **&x**) != **EOF**) { ... }

- **NULL**: já usada mais do que uma vez, representa um **ponteiro que não aponta para lado nenhum**. Pode ser encontrada, em geral, em '**stdio.h**' ou noutra file por ela chamada. A sua definição é feita por:

  #**define NULL** ((**void** *) **0**)