

12ª Aula - Variáveis e Constantes.

Programação

Mestrado em Engenharia Física Tecnológica

Samuel M. Eleutério
sme@tecnico.ulisboa.pt

Departamento de Física
Instituto Superior Técnico
Universidade de Lisboa

Tipos Básicos

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

Variáveis de tipo inteiro:

- **char**: ocupa **1 byte** ($8 \text{ bits} = 2^8 = 256$) de memória. Pode ser usado para guardar um caracter ou um inteiro de 1 byte;
- **int**: ocupa **4 bytes** ($4 \times 8 = 32 \text{ bits} = 2^{32} = 4\,294\,967\,296$) 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** ou **8 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**'.

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 registrar 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 constituírem 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 **codificações específicas** 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, hexadecimal 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 character**. Exemplos: a, b, **'\n'** (nova linha), **'%%'** (percentagem), **'\\'** (traço para trás), **'\"'** (aspas), 0, 1, etc.
- De **strings** (cadeias (vectores) 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 um ficheiro** (inglês: **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**. A sua definição é feita por:

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


Variáveis Constantes

- Apesar de parecer uma **contradição** este conceito existe em **C** e tem um **significado** bem preciso.
- Vejamos em que contextos de usa e quais as suas implicações.
- Por vezes queremos que **certas variáveis** se mantenham **constantes** durante a execução de um programa.
- E, se por algum erro, tal não acontecer, o programa não permita a sua alteração.
- Há duas maneiras de impor um valor constante:
 - Usando a instrução '**#define**' do **pré-processor**. Note-se que, neste caso, de facto não se tem uma variável (ver mais à frente);
 - Usando o qualificativo **const**.

Variáveis Constantes - '#define' ('Prog21_01.c')

- Quando se usa a instrução 'define' do pré-processador, exemplo:

```
#define NPONTOS 25
```

ele vai procurar as ocorrências de 'NPONTOS' e substituí-las pelo seu valor '25'. Assim, quando o compilador vai compilar o programa já lá não encontra 'NPONTOS' mas o seu valor: '25'.

- Tudo se comporta como se mandássemos substituir em todo o programa 'NPONTOS' por '25' e depois o compilássemos.
- Se quisermos declarar uma variável dimensionada com aquele valor, podemos fazer simplesmente:

```
double curva[NPONTOS] ;
```

- De modo idêntico, num ciclo que preencha esses valores, faríamos
for (i = 0 ; i < NPONTOS ; ++i) ...
- Note-se que se desejássemos alterar noutro momento o número de pontos a ler, bastava alterar o seu valor no 'define'.

Variáveis Constantes - 'const' ('Prog21_01.c')

- O qualificativo 'const' é interpretado pelo compilador e aplica-se em conjunção com os tipos.
- O valor de uma **variável**, marcada como 'const', uma vez fixado **não pode ser alterado**.
- Note-se que, o modo como se aplica, determina o ser feito:
 - **const tipo i2**;
O seu valor não pode ser alterado.
 - **const tipo *x1 = constante** ;
O valor apontado por 'x1' é **constante** mas, o seu ponteiro pode ser **alterado**.
 - **tipo *const x2 = constante** ;
O ponteiro 'x2' é **constante**, no entanto, o valor por ele apontado pode ser **alterado**.
 - **const tipo *const x3 = constante** ;
Tanto o ponteiro como o valor por ele apontado **não podem ser alterados**.

Tempo de Vida das Variáveis (I)

- Um dos aspectos importantes quando definimos uma **variável** é sabermos **onde** ela fica definida.
- Se declaramos uma **variável** numa **função**, ou num **bloco encastrado** no interior de uma função, ela apenas **existe durante** a execução da zona em que foi definida, isto é, na função ou no bloco. Diremos que ela é uma **variável local**.
- Se em vez disso declararmos uma **variável** num ficheiro, fora das funções, ela terá uma **existência global** em **todas** as funções que se encontram definidas depois dela. E diremos que ela é uma **variável global**.
- Note-se que uma variável pode ser **global** num **ficheiro** e **local** num **programa**, pois, só existe nesse ficheiro.
- Existem diversos tipos de **qualificadores** que caracterizam o **comportamento** temporal das variáveis.

Tempo de Vida das Variáveis - Tipos

Tipo: auto

- É o tipo a que correspondem as variáveis quando *mais nada é dito*. Têm do ponto de vista do programa sempre **vida local** em blocos, em funções ou em ficheiros.
- Uma variável local **é reinicializada** a cada entrada na zona do código em que está definida e só é definida a partir do momento em que é declarada.
- Quando há duas variáveis com o **mesmo nome** definidas em **níveis diferentes**, a que prevalece é a do **nível mais baixo**.

Tempo de Vida das Variáveis - Tipos ('Prog21_02.c')

Tipo: **static**

- É semelhante ao anterior mas as variáveis **não são inicializadas** a cada entrada na zona do código. Por exemplo se declararmos **dentro** de uma função:

```
static int numero = 10 ;
```

a atribuição do valor ('10') é feita **apenas** na primeira vez chamada, a partir daí, sempre que for alterada o novo valor **permanecerá** até à chamada seguinte.

- Quando é declarada num **ficheiro** indica que só nele é conhecida.

Tempo de Vida das Variáveis - Tipos

Tipo: **extern**

- Uma variável global num ficheiro pode ser utilizada noutros ficheiros desde que seja considerada como **extern** nestes.

Exemplo:

```
extern int versao ;
```

- Note-se que ela apenas poderá ser **declarada uma só vez**. Nas restantes é declarada como **extern**.
- Em geral, essa declaração não é feita explicitamente no ficheiros em que vai ser usada. Em vez disso, a declaração de **extern** é feita num ficheiro **'.h'** que irá ser incluído nesses ficheiros.

Tempo de Vida das Variáveis - Tipos

Tipo: register

- Quando um programa é executado as suas variáveis são colocadas na memória.
- No entanto, o CPU tem algumas **posições de memória** que se chamam **registos** de acesso muito mais rápido.
- A indicação de **register** dirá que é uma variável muito usada e que deve ser colocada num registo. Mas o sistema decidirá se o deve fazer ou não.
- Só se aplica a variáveis de **tipo simples** (isto é, não se aplica a variáveis dimensionadas nem a estrutura).

Tempo de Vida das Variáveis - Tipos

Tipo: **volatile**

- Esta declaração usa-se quando o valor de uma variável pode ser alterado de *forma não controlada* pelo programa.
- Este tipo de situações ocorre quando se tem um programa que **trata certos valores**, por exemplo, de um hardware **externo**, e esses valores **podem mudar** devido a esse **hardware** e não devido ao programa.
- Num ciclo, por exemplo, a **otimização** pode colocar esses valores num **registo**. Nesse caso, o que o programa leria era o que estava no **registo** e não os valores dados pelo **hardware**.
- Com a declaração de **volatile** o programa vai obrigatoriamente testar os valores nas posições estabelecidas.

Organização da Memória

Um **programa**, ao ser executado, é organizado, em termo de **armazenamento na memória**, do seguinte modo:

- **Segmento de código**

Onde se encontra o **código** propriamente dito do programa;

- **Stack (pilha)**

Onde estão as **variáveis locais**. Vai variando à medida que as funções vão sendo chamadas;

- **Segmento de dados**

Onde vão estar as **variáveis globais** do programa;

- **Heap**

Onde irá ficar a **memória alocada dinamicamente**. Varia de acordo com as instruções de alocação e libertação de memória. Está acessível a todo o programa.

Operações básicas ('Prog22_0(1a5).c')

- Os **operadores**, de acordo com o **número de entidades** sobre os quais actuam, **classificam-se** em:
 - **Unários**: quando actuam sobre **um** elemento.
Exemplos: negação, '++', etc..
 - **Binários**: quando actuam sobre **dois** elementos.
Exemplos: maior, '+', '*', etc..
 - **Ternários**: quando actuam sobre **três** elementos.
Exemplo: '? :' (**Condição ? Se_Verdadeiro : Se_Falso**)
- As operações podem ainda **classificar-se** tendo em conta o **modo como operam** em:
 - Operações **aritméticas**;
 - Operações **sobre bits**;
 - Operações **sobre ponteiros**;
 - Operações **lógicas relacionais**.

Operações Aritméticas - Unárias ('Prog22_01.c')

As principais **operações** aritméticas **unárias** são:

- **'++'**: $++i, i++ \Leftrightarrow i = i + 1 \Leftrightarrow i += 1$;
- **'--'**: $--i, i-- \Leftrightarrow i = i - 1 \Leftrightarrow i -= 1$;
- **'-'** (Troca de sinal): $j = -i$;

Nota: Não esquecer que no caso dos operadores **'++'** e **'--'**, a posição em que se encontram determina o momento em que são executados.

- Suas utilizações múltiplas podem tem resultados imprevisíveis.
- Devem ser usados com um certo cuidado.

Operações Aritméticas - Binárias ('Prog22_02.c')

As principais **operações** aritméticas **binárias** são:

- **Soma** '+';
- **Subtração** '-';
- **Multiplicação** '*';
- **Divisão** '/';
- **Módulo** (resto da divisão para inteiros) '%'.

Notas:

- Quando se efectuam operações entre variáveis de **tipos diferentes** a norma do **C** define como essas conversões automáticas devem ser feitas.
- Em geral, pode dizer-se que se convertem os valores para o tipo mais abrangente de entre os dois valores em causa.
- **Ver regras de conversão na bibliografia.**

Operações Aritméticas - Binárias Compostas

As principais **operações** aritméticas **binárias compostas** são:

- **'+='**: `'x += y;'` \Leftrightarrow `'x = x + y;'`
- **'-='**: `'x -= y'` \Leftrightarrow `'x = x - y;'`
- **'*='**: `'x *= y'` \Leftrightarrow `'x = x * y;'`
- **'/='**: `'x /= y'` \Leftrightarrow `'x = x / y;'`
- **'%='**: `'x %= y'` \Leftrightarrow `'x = x % y;'`

Pequenas distrações na escrita podem conduzir a algumas situações delicadas:

- A instrução **'a=*b'** significa que estamos a guardar em **'a'** o valor do ponteiro **'b'**;
- A instrução **'a=b/*c'** significa que iniciámos um **comentário!** E não que estamos a dividir **'b'** pelo valor do ponteiro **'c'**. Evita-se esta situação escrevendo **'a=b/ *c'** ou **'a=b/(*c)'**.

Operações Lógicas ('Prog22_03.c')

As **operações lógicas**, são instruções capazes de retornar valores **verdadeiros** ou **falsos**. Em **C**, qualquer inteiro diferente de zero é verdadeiro. As principais **operações lógicas** são:

- **Unário** – Negação: **'!a'**;
- **Binários** – Igualdade: **'a==b'**; Diferença: **'a!=b'**; Maiores, menores, etc.: **'a<b'**, **'a<=b'**, **'a>b'**, **'a>=b'**; Conjunção (\wedge): **'a&&b'**; Disjunção (\vee): **'a||b'**;
- **Ternário** – **'(Condição ? Se_Verdadeira : Se_Falso)'**.

Exemplo com o cálculo do módulo de **'a'**:

$$x = (a \geq 0 ? a : -a);$$

Este operador tem **três** argumentos: o **primeiro** é uma operação susceptível de ser verdadeira ou falsa, o **segundo** é o valor a atribuir se o **primeiro** argumento for **verdadeiro** e o **terceiro** é o valor a atribuir se o **primeiro** argumento for **falso**.

Operações sobre Bits (I) ('Prog22_04.c')

- Iremos aqui estudar os **operadores** que actuam sobre a **representação binárias** de números, isto é, sobre os '**bits**'.
- Os operadores **binários** que actuam os **bits** são: **Conjunção** (AND) '&', **Disjunção** (OR) '|', **Disjunção exclusiva** (XOR) '^', **Deslocamento à direita** (>>) e à **esquerda** (<<) e o **Complemento** (~).
- Podemos ver um exemplo, para 8 bits, das operações **Conjunção** (**AND**, '&'), **Disjunção** (**OR**, '|'), **Disjunção exclusiva** (**XOR**, '^'), com $00101010 = 42_{10} = 0x2a_{16}$ e $00010010 = 18_{10} = 0x12_{16}$:

$$\begin{array}{r} 00101010 \\ \& 00010010 \\ \hline 00000010 = 2_{10} \end{array}$$

$$\begin{array}{r} 00101010 \\ | 00010010 \\ \hline 00111010 = 58_{10} \end{array}$$

$$\begin{array}{r} 00101010 \\ ^ 00010010 \\ \hline 00111000 = 56_{10} \end{array}$$

Operações sobre Bits (II) ('Prog22_04.c')

- Os operadores **deslocamento à direita** (\gg) e **à esquerda** (\ll), como o nome indica fazem uma translação dos bits para a direita ou para a esquerda.
- Assim, deslocar o valor **'2'** para a esquerda corresponde a passar `'...0010'` para `'...0100'`. Ou seja passamos **'2'** para **'4'**.
- O deslocamento para a direita corresponde à operação inversa.
- Note-se que quando temos **'1'** e deslocamos para a direita ficamos com **'0'**. O mesmo se passa quando passamos o último bit à esquerda para a esquerda.
- Por isso, muita atenção às **situações limites** em que se fazem **"desaparecer"** bits.
- O **Complemento** (\sim) inverte os valores de todos os bits em questão, assim:

$$11010010 (-46_{10}) = \sim 00101101 (45_{10})$$

Operações sobre Bits - Operadores Compostos

- Também aqui podemos utilizar, à semelhança do que se fez para as operações aritméticas básicas, a composição e atribuição de novo valor:

Operação	Sequência Equivalente
$x \&= y$	$x = x \& y$
$x = y$	$x = x y$
$x \hat{=} y$	$x = x \hat{ } y$
$x \gg= y$	$x = x \gg y$
$x \ll= y$	$x = x \ll y$

Operador Vírgula (',')

- É um operador que só aparece em algumas situações específicas.
- Note-se que não nos estamos a referir à **vírgula** que aparece a separar variáveis.
- Este operador aparece em situações como **ciclo 'for'** e serve para **separar instruções**:

```
for (i = 0, j = 0 ; i < 15 ; ++i, ++j) { ... }
```

Precedência das Operações

Prioridade	Operador	Associatividade	Tipo
1	() [] -> .	→	
2	! ~ ++ -- - + (molde) * & sizeof	←	unário
3	* / %	→	binário
4	+ -	→	binário
5	<< >>	→	binário
6	< <= > >=	→	binário
7	== !=	→	binário
8	&	→	binário
9	^	→	binário
10		→	binário
11	&&	→	binário
12		→	binário
13	? :	←	ternário
14	= e compostos	←	binário
15	,	→	binário