10ª Aula - Operadores de Molde ('Casting'). Atribuição de Memória. Ponteiros. Enumerados.

Programação

Mestrado em Engenharia Física Tecnológica

Samuel M. Eleutério
sme@tecnico.ulisboa.pt

Departamento de Física
Instituto Superior Técnico
Universidade de Lisboa

Operadores de molde ('casting')

- Já vimos, mais de uma vez, como se utilizam os **operadores de molde** (**casting**): eles permitiram-nos fazer conversões entre **tipos compatíveis**.

Operadores de molde ('casting')

- Já vimos, mais de uma vez, como se utilizam os **operadores de molde** (**casting**): eles permitiram-nos fazer conversões entre **tipos compatíveis**.
- Usámo-los, por exemplo, para gerar números aleatórios no intervalo [0,1], isto é, **converter inteiros em reais**.

Operadores de molde ('casting')

- Já vimos, mais de uma vez, como se utilizam os **operadores de molde** (**casting**): eles permitiram-nos fazer conversões entre **tipos compatíveis**.
- Usámo-los, por exemplo, para gerar números aleatórios no intervalo [0,1], isto é, **converter inteiros em reais**.
- No entanto, há um **outro papel** muito importante por eles desempenhado.

Operadores de molde ('casting')

- Já vimos, mais de uma vez, como se utilizam os **operadores de molde** (**casting**): eles permitiram-nos fazer conversões entre **tipos compatíveis**.
- Usámo-los, por exemplo, para gerar números aleatórios no intervalo [0,1], isto é, **converter inteiros em reais**.
- No entanto, há um **outro papel** muito importante por eles desempenhado.
- Vamos ver nos pontos seguintes como se usa o ponteiro '**void ***' quando ainda **não se sabe o tipo** para que irá apontar.

Operadores de molde ('casting')

- Já vimos, mais de uma vez, como se utilizam os **operadores de molde** (**casting**): eles permitiram-nos fazer conversões entre **tipos compatíveis**.
- Usámo-los, por exemplo, para gerar números aleatórios no intervalo [0,1], isto é, **converter inteiros em reais**.
- No entanto, há um **outro papel** muito importante por eles desempenhado.
- Vamos ver nos pontos seguintes como se usa o ponteiro '**void ***' quando ainda **não se sabe o tipo** para que irá apontar.
- Mas chega sempre o momento em que o tipo **tem de ser especificado** e, nessa altura, devemos usar o **operador** respectivo para atribuir o **tipo correcto**.

Operadores de molde ('casting')

- Já vimos, mais de uma vez, como se utilizam os **operadores de molde** (**casting**): eles permitiram-nos fazer conversões entre **tipos compatíveis**.
- Usámo-los, por exemplo, para gerar números aleatórios no intervalo [0,1], isto é, **converter inteiros em reais**.
- No entanto, há um **outro papel** muito importante por eles desempenhado.
- Vamos ver nos pontos seguintes como se usa o ponteiro '**void ***' quando ainda **não se sabe o tipo** para que irá apontar.
- Mas chega sempre o momento em que o tipo **tem de ser especificado** e, nessa altura, devemos usar o **operador** respectivo para atribuir o **tipo correcto**.
- Nos pontos seguintes vão mostrar-se situações que **exigem** a sua **utilização**.

Atribuição de Memória ('Prog12_01.c')

- Um operador particularmente importante para **gestão de memória** é o operador:

Atribuição de Memória ('Prog12_01.c')

- Um operador particularmente importante para **gestão de memória** é o operador:

  sizeof (**nome_do_tipo**);

Atribuição de Memória ('Prog12_01.c')

- Um operador particularmente importante para **gestão de memória** é o operador:

  sizeof (**nome_do_tipo**);

  ele retorna o **comprimento**, em bytes, do **tipo** pedido.

Atribuição de Memória ('Prog12_01.c')

- Um operador particularmente importante para **gestão de memória** é o operador:

  sizeof (**nome_do_tipo**);

  ele retorna o **comprimento**, em bytes, do **tipo** pedido.
- **Nota:** os **parêntesis** só são obrigatórios se o argumento for o **nome de um tipo**. Falaremos disso mais tarde.

Atribuição de Memória ('Prog12_01.c')

- Um operador particularmente importante para **gestão de memória** é o operador:

  sizeof (**nome_do_tipo**);

  ele retorna o **comprimento**, em bytes, do **tipo** pedido.
- **Nota:** os **parêntesis** só são obrigatórios se o argumento for o **nome de um tipo**. Falaremos disso mais tarde.
- Admitamos que necessitamos de **guardar** reais num vector mas **não sabemos** à partida qual o seu comprimento. Que fazemos?

Atribuição de Memória ('Prog12_01.c')

- Um operador particularmente importante para **gestão de memória** é o operador:

  sizeof (**nome_do_tipo**);

  ele retorna o **comprimento**, em bytes, do **tipo** pedido.
- **Nota:** os **parêntesis** só são obrigatórios se o argumento for o **nome de um tipo**. Falaremos disso mais tarde.
- Admitamos que necessitamos de **guardar** reais num vector mas **não sabemos** à partida qual o seu comprimento. Que fazemos?
- Podemos fazer, como anteriormente, sobredimensionar o vector e muito provavelmente não ter memória suficiente ou

Atribuição de Memória ('Prog12_01.c')

- Um operador particularmente importante para **gestão de memória** é o operador:

  sizeof (**nome_do_tipo**);

  ele retorna o **comprimento**, em bytes, do **tipo** pedido.
- **Nota:** os **parêntesis** só são obrigatórios se o argumento for o **nome de um tipo**. Falaremos disso mais tarde.
- Admitamos que necessitamos de **guardar** reais num vector mas **não sabemos** à partida qual o seu comprimento. Que fazemos?
- Podemos fazer, como anteriormente, sobredimensionar o vector e muito provavelmente não ter memória suficiente ou
- Podemos criá-lo quando necessário e com o tamanho exigido.

Atribuição de Memória ('Prog12_01.c')

- Um operador particularmente importante para **gestão de memória** é o operador:

  sizeof (**nome_do_tipo**);

  ele retorna o **comprimento**, em bytes, do **tipo** pedido.
- **Nota:** os **parêntesis** só são obrigatórios se o argumento for o **nome de um tipo**. Falaremos disso mais tarde.
- Admitamos que necessitamos de **guardar** reais num vector mas **não sabemos** à partida qual o seu comprimento. Que fazemos?
- Podemos fazer, como anteriormente, sobredimensionar o vector e muito provavelmente não ter memória suficiente ou
- Podemos criá-lo quando necessário e com o tamanho exigido.
- Para tal usamos a função **malloc**:

  void ***malloc** (**size_t size**);

Atribuição de Memória (II)
('Prog12_02/3.c' e 'Prog08_11.c' [ver 08_07])

- Se quisermos criar um vector de 50 **doubles**, declaramos um ponteiro para **double**, **reservamos** (**alloc**) o **espaço de memória** desejado e **apontamos** para lá o ponteiro:

Atribuição de Memória (II)
('Prog12_02/3.c' e 'Prog08_11.c' [ver 08_07])

- Se quisermos criar um vector de 50 **double**s, declaramos um ponteiro para **double**, **reservamos** (**alloc**) o **espaço de memória** desejado e **apontamos** para lá o ponteiro:

```
double *p ;
p = (double *) malloc (50 * sizeof (double));
```

Atribuição de Memória (II)
('Prog12_02/3.c' e 'Prog08_11.c' [ver 08_07])

- Se quisermos criar um vector de 50 **double**s, declaramos um ponteiro para **double**, **reservamos** (**alloc**) o **espaço de memória** desejado e **apontamos** para lá o ponteiro:

```
double *p ;
p = (double *) malloc (50 * sizeof (double));
```

- Se desejarmos **alterar o espaço de memória** atribuído a um ponteiro, podemos fazê-lo usando a função **realloc**:

Atribuição de Memória (II)
('Prog12_02/3.c' e 'Prog08_11.c' [ver 08_07])

- Se quisermos criar um vector de 50 **double**s, declaramos um ponteiro para **double**, **reservamos** (**alloc**) o **espaço de memória** desejado e **apontamos** para lá o ponteiro:

```
double *p ;
p = (double *) malloc (50 * sizeof (double));
```

- Se desejarmos **alterar o espaço de memória** atribuído a um ponteiro, podemos fazê-lo usando a função **realloc**:

```
void *realloc (void *ptr, size_t size);
```

Atribuição de Memória (II)
('Prog12_02/3.c' e 'Prog08_11.c' [ver 08_07])

- Se quisermos criar um vector de 50 **double**s, declaramos um ponteiro para **double**, **reservamos** (**alloc**) o **espaço de memória** desejado e **apontamos** para lá o ponteiro:

```
double *p ;
p = (double *) malloc (50 * sizeof (double));
```

- Se desejarmos **alterar o espaço de memória** atribuído a um ponteiro, podemos fazê-lo usando a função **realloc**:

```
void *realloc (void *ptr, size_t size);
```

- Assim, se desejarmos alterar o seu tamanho de 50 para 70:

Atribuição de Memória (II)
('Prog12_02/3.c' e 'Prog08_11.c' [ver 08_07])

- Se quisermos criar um vector de 50 **double**s, declaramos um ponteiro para **double**, **reservamos** (**alloc**) o **espaço de memória** desejado e **apontamos** para lá o ponteiro:

```
double *p ;
p = (double *) malloc (50 * sizeof (double));
```

- Se desejarmos **alterar o espaço de memória** atribuído a um ponteiro, podemos fazê-lo usando a função **realloc**:

```
void *realloc (void *ptr, size_t size);
```

- Assim, se desejarmos alterar o seu tamanho de 50 para 70:

```
p = (double *) realloc (p, 70 * sizeof (double));
```

Atribuição de Memória (II) ('Prog12_02/3.c' e 'Prog08_11.c' [ver 08_07])

- Se quisermos criar um vector de 50 **double**s, declaramos um ponteiro para **double**, **reservamos** (**alloc**) o **espaço de memória** desejado e **apontamos** para lá o ponteiro:

  double ***p** ;

  p = (**double** *) **malloc** (50 * **sizeof** (**double**));

- Se desejarmos **alterar o espaço de memória** atribuído a um ponteiro, podemos fazê-lo usando a função **realloc**:

  void ***realloc** (**void** ***ptr**, **size_t size**);

- Assim, se desejarmos alterar o seu tamanho de 50 para 70:

  p = (**double** *) **realloc** (**p**, 70 * **sizeof** (**double**));

- Finalmente, existe uma outra função que **liberta** o espaço reservado pela aplicação das funções **malloc** ou **realloc**. Essa função chama-se **free**:

Atribuição de Memória (II)
('Prog12_02/3.c' e 'Prog08_11.c' [ver 08_07])

- Se quisermos criar um vector de 50 **double**s, declaramos um ponteiro para **double**, **reservamos** (**alloc**) o **espaço de memória** desejado e **apontamos** para lá o ponteiro:

```
double *p ;
p = (double *) malloc (50 * sizeof (double));
```

- Se desejarmos **alterar o espaço de memória** atribuído a um ponteiro, podemos fazê-lo usando a função **realloc**:

```
void *realloc (void *ptr, size_t size);
```

- Assim, se desejarmos alterar o seu tamanho de 50 para 70:

```
p = (double *) realloc (p, 70 * sizeof (double));
```

- Finalmente, existe uma outra função que **liberta** o espaço reservado pela aplicação das funções **malloc** ou **realloc**. Essa função chama-se **free**:

```
void free (void *ptr);
```

Ponteiros ('Prog18_01.c')

- Em **C** todas as variáveis podem ser encaradas de dois pontos de vista diferentes:

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;

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**, isto é, pelo conteúdo da sua localização.

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**, isto é, pelo conteúdo da sua localização.
- As variáveis que guardam a **localização na memória** do espaço atribuído às variáveis (**endereço**) chamam-se **ponteiros**.

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**, isto é, pelo conteúdo da sua localização.
- 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:

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**, isto é, pelo conteúdo da sua localização.
- 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**. etc. ;

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**, isto é, pelo conteúdo da sua localização.
- 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**. etc. ;
- Se há **ponteiros**, então também deverão existir **ponteiros** que **apontam** para **ponteiros**!:

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**, isto é, pelo conteúdo da sua localização.
- 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**. etc. ;
- Se há **ponteiros**, então também deverão existir **ponteiros** que **apontam** para **ponteiros**!:

  double ****r** ;

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**, isto é, pelo conteúdo da sua localização.
- 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. etc. ;
- Se há **ponteiros**, então também deverão existir **ponteiros** que **apontam** para **ponteiros**!:

  double **r ;

  e assim sucessivamente... **double ***s**, **int ****v**, 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).

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.

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:

'.

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.};

  '.

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**[ ];'.

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:

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}};

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.

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 ***'.

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 '**void ***' pode assumir o **valor** dum ponteiro de qualquer outro tipo, e, inversamente, pode ser atribuído a **qualquer outro ponteiro**.

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 '**void ***' pode assumir o **valor** dum 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:

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 '**void ***' pode assumir o **valor** dum 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;

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 '**void ***' pode assumir o **valor** dum 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;

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 '**void ***' pode assumir o **valor** dum 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) ('Prog18_02.c')

- Para além dos operadores atrás referidos são possíveis as **operações** de **soma** e **subtração**.

Ponteiros (IV) ('Prog18_02.c')

- 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. E, se for '**n**':

Ponteiros (IV) ('Prog18_02.c')

- 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. E, se for '**n**':

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

Ponteiros (IV) ('Prog18_02.c')

- 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. E, se for '**n**':

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

  ou, inversamente,

Ponteiros (IV) ('Prog18_02.c')

- 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. E, se for '**n**':

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

  ou, inversamente,

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

Ponteiros (IV) ('Prog18_02.c')

- 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. E, se for '**n**':

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

  ou, inversamente,

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

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

Ponteiros (IV) ('Prog18_02.c')

- 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. E, se for '**n**':

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

  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**);

Ponteiros (IV) ('Prog18_02.c')

- 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. E, se for '**n**':

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

  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 '**++**', '**− −**', '**+=**', '**−=**'.

Ponteiros (IV) ('Prog18_02.c')

- 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. E, se for '**n**':

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

  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 '++', '−−', '+=', '−='.
- Actualmente, na maioria dos sistemas os **ponteiros** usam **8 bytes**, embora em sistemas mais antigos ainda se usem **4 bytes**.

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**.

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**'.

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.

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.

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.