

8ª Aula - Ordenação.

Programação

Mestrado em Engenharia Física Tecnológica

Samuel M. Eleutério

sme@tecnico.ulisboa.pt

Departamento de Física
Instituto Superior Técnico
Universidade de Lisboa

Ordenação - Introdução


- **Ordenar valores**, quer **numéricos** quer **literais**, é uma tarefa frequente em muitos tipos de programas.

Ordenação - Introdução

- **Ordenar valores**, quer **numéricos** quer **literais**, é uma tarefa frequente em muitos tipos de programas.
- Por vezes, o número de elementos a ordenar é **bastante grande** o que obriga a procurar **estratégias eficientes de ordenação**.


Ordenação - Introdução

- **Ordenar valores**, quer **numéricos** quer **literais**, é uma tarefa frequente em muitos tipos de programas.
- Por vezes, o número de elementos a ordenar é **bastante grande** o que obriga a procurar **estratégias eficientes de ordenação**.
- O algoritmo mais simples de ordenação (embora pouco eficiente) designa-se por **bubble sort**¹ e consiste no seguinte (crescente):

¹Porque o método se assemelha às bolhas (bubble) a subirem nos líquidos 


Ordenação - Introdução

- **Ordenar valores**, quer **numéricos** quer **literais**, é uma tarefa frequente em muitos tipos de programas.
- Por vezes, o número de elementos a ordenar é **bastante grande** o que obriga a procurar **estratégias eficientes de ordenação**.
- O algoritmo mais simples de ordenação (embora pouco eficiente) designa-se por **bubble sort**¹ e consiste no seguinte (crescente):
 - 1 Começa-se nos **dois primeiros elementos**, se estiverem na **ordem certa**, **ficam** como estão; caso contrário **trocam-se**;

¹Porque o método se assemelha às bolhas (bubble) a subirem nos líquidos 


Ordenação - Introdução

- **Ordenar valores**, quer **numéricos** quer **literais**, é uma tarefa frequente em muitos tipos de programas.
- Por vezes, o número de elementos a ordenar é **bastante grande** o que obriga a procurar **estratégias eficientes de ordenação**.
- O algoritmo mais simples de ordenação (embora pouco eficiente) designa-se por **bubble sort**¹ e consiste no seguinte (crescente):
 - 1 Começa-se nos **dois primeiros elementos**, se estiverem na **ordem certa**, **ficam** como estão; caso contrário **trocam-se**;
 - 2 Depois passa-se ao **segundo** e **terceiro** **repete-se a operação**;

¹Porque o método se assemelha às bolhas (bubble) a subirem nos líquidos 


Ordenação - Introdução

- **Ordenar valores**, quer **numéricos** quer **literais**, é uma tarefa frequente em muitos tipos de programas.
- Por vezes, o número de elementos a ordenar é **bastante grande** o que obriga a procurar **estratégias eficientes de ordenação**.
- O algoritmo mais simples de ordenação (embora pouco eficiente) designa-se por **bubble sort**¹ e consiste no seguinte (crescente):
 - 1 Começa-se nos **dois primeiros elementos**, se estiverem na **ordem certa**, **ficam** como estão; caso contrário **trocam-se**;
 - 2 Depois passa-se ao **segundo** e **terceiro** **repete-se a operação**;
 - 3 E assim sucessivamente até ao **último par**;

¹Porque o método se assemelha às bolhas (bubble) a subirem nos líquidos 


Ordenação - Introdução

- **Ordenar valores**, quer **numéricos** quer **literais**, é uma tarefa frequente em muitos tipos de programas.
- Por vezes, o número de elementos a ordenar é **bastante grande** o que obriga a procurar **estratégias eficientes de ordenação**.
- O algoritmo mais simples de ordenação (embora pouco eficiente) designa-se por **bubble sort**¹ e consiste no seguinte (crescente):
 - 1 Começa-se nos **dois primeiros elementos**, se estiverem na **ordem certa**, **ficam** como estão; caso contrário **trocam-se**;
 - 2 Depois passa-se ao **segundo** e **terceiro** repete-se a **operação**;
 - 3 E assim sucessivamente até ao **último par**;
 - 4 Se **nenhuma troca foi feita acabou**, **senão** volta-se ao **início**.

¹Porque o método se assemelha às bolhas (bubble) a subirem nos líquidos 


Ordenação - Introdução

- **Ordenar valores**, quer **numéricos** quer **literais**, é uma tarefa frequente em muitos tipos de programas.
- Por vezes, o número de elementos a ordenar é **bastante grande** o que obriga a procurar **estratégias eficientes de ordenação**.
- O algoritmo mais simples de ordenação (embora pouco eficiente) designa-se por **bubble sort**¹ e consiste no seguinte (crescente):
 - 1 Começa-se nos **dois primeiros elementos**, se estiverem na **ordem certa**, **ficam** como estão; caso contrário **trocam-se**;
 - 2 Depois passa-se ao **segundo** e **terceiro** **repete-se a operação**;
 - 3 E assim sucessivamente até ao **último par**;
 - 4 Se **nenhuma troca foi feita acabou**, **senão** volta-se ao **início**.
- Se tivermos **N** elementos a ordenar, na pior das hipóteses, teremos de efectuar quase **N²** comparações. O que é um número de operações bastante elevado para ordenar 1 milhão de valores.

¹Porque o método se assemelha às bolhas (bubble) a subirem nos líquidos 

Ordenação - Introdução

- **Ordenar valores**, quer **numéricos** quer **literais**, é uma tarefa frequente em muitos tipos de programas.
- Por vezes, o número de elementos a ordenar é **bastante grande** o que obriga a procurar **estratégias eficientes de ordenação**.
- O algoritmo mais simples de ordenação (embora pouco eficiente) designa-se por **bubble sort**¹ e consiste no seguinte (crescente):
 - 1 Começa-se nos **dois primeiros elementos**, se estiverem na **ordem certa**, **ficam** como estão; caso contrário **trocam-se**;
 - 2 Depois passa-se ao **segundo** e **terceiro** **repete-se a operação**;
 - 3 E assim sucessivamente até ao **último par**;
 - 4 Se **nenhuma troca foi feita acabou**, **senão** volta-se ao **início**.
- Se tivermos **N** elementos a ordenar, na pior das hipóteses, teremos de efectuar quase **N²** comparações. O que é um número de operações bastante elevado para ordenar 1 milhão de valores.
- Um dos algoritmos de ordenação mais eficientes é o **quick sort**.

¹Porque o método se assemelha às bolhas (bubble) a subirem nos líquidos 

Ordenação (método de bubble sort) - (Prog08_05.c)

Seja um vector **r1** que se pretende ordenar por **ordem crescente**

Ordenação (método de bubble sort) - (Prog08_05.c)

Seja um vector **r1** que se pretende ordenar por **ordem crescente**

```
while (1)  
{
```

■ **Ciclo de ordenação;**

```
}
```

Ordenação (método de bubble sort) - (Prog08_05.c)

Seja um vector **r1** que se pretende ordenar por **ordem crescente**

```
while (1)
{
    t = 0;
```

- **Ciclo de ordenação;**
- Inicializa-se o **teste de troca;**

```
}
```

Ordenação (método de bubble sort) - (Prog08_05.c)

Seja um vector **r1** que se pretende ordenar por **ordem crescente**

```
while (1)
```

```
{
```

```
  t = 0;
```

```
  for (i1=0 ; i1<qt - 1 ;++i1)
```

```
    {
```

```
    }
```

```
}
```

- **Ciclo de ordenação;**
- Inicializa-se o **teste de troca;**
- Vai percorrer-se o vector para testar eventuais trocas;

Ordenação (método de bubble sort) - (Prog08_05.c)

Seja um vector **r1** que se pretende ordenar por **ordem crescente**

```
while (1)
{
    t = 0;
    for (i1=0 ; i1<qt - 1 ;++i1)
    {
        if (r1[i1] > r1[i1+1])
        {

        }
    }
}
```

- **Ciclo de ordenação**;
- Inicializa-se o **teste de troca**;
- Vai percorrer-se o vector para testar eventuais trocas;
- Se $r1[i1] > r1[i1 + 1]$ vão **trocar-se** os seus valores;

Ordenação (método de bubble sort) - (Prog08_05.c)

Seja um vector **r1** que se pretende ordenar por **ordem crescente**

```
while (1)
{
    t = 0;
    for (i1=0 ; i1<qt - 1 ;++i1)
    {
        if (r1[i1] > r1[i1+1])
        {
            r0 = r1[i1];
        }
    }
}
```

- **Ciclo de ordenação**;
- Inicializa-se o **teste de troca**;
- Vai percorrer-se o vector para testar eventuais trocas;
- Se $r1[i1] > r1[i1 + 1]$ vão **trocar-se** os seus valores;
- Para trocar dois valores, usamos uma **variável intermédia**,

Ordenação (método de bubble sort) - (Prog08_05.c)

Seja um vector **r1** que se pretende ordenar por **ordem crescente**

```
while (1)
{
    t = 0;
    for (i1=0 ; i1<qt - 1 ;++i1)
    {
        if (r1[i1] > r1[i1+1])
        {
            r0 = r1[i1];
            r1[i1] = r1[i1+1];
            r1[i1+1] = r0;
        }
    }
}
```

- **Ciclo de ordenação**;
- Inicializa-se o **teste de troca**;
- Vai percorrer-se o vector para testar eventuais trocas;
- Se $r1[i1] > r1[i1 + 1]$ vão **trocar-se** os seus valores;
- Para trocar dois valores, usamos uma **variável intermédia**, depois trocam-se os valores;

Ordenação (método de bubble sort) - (Prog08_05.c)

Seja um vector **r1** que se pretende ordenar por **ordem crescente**

```
while (1)
{
    t = 0;
    for (i1=0 ; i1<qt - 1 ;++i1)
    {
        if (r1[i1] > r1[i1+1])
        {
            r0 = r1[i1];
            r1[i1] = r1[i1+1];
            r1[i1+1] = r0;
            t = 1;
        }
    }
}
```

- **Ciclo de ordenação**;
- Inicializa-se o **teste de troca**;
- Vai percorrer-se o vector para testar eventuais trocas;
- Se $r1[i1] > r1[i1 + 1]$ vão **trocar-se** os seus valores;
- Para trocar dois valores, usamos uma **variável intermédia**, depois trocam-se os valores;
- Havendo troca marca-se '**t=1**';

Ordenação (método de bubble sort) - (Prog08_05.c)

Seja um vector **r1** que se pretende ordenar por **ordem crescente**

```
while (1)
{
    t = 0;
    for (i1=0 ; i1<qt - 1 ;++i1)
    {
        if (r1[i1] > r1[i1+1])
        {
            r0 = r1[i1];
            r1[i1] = r1[i1+1];
            r1[i1+1] = r0;
            t = 1;
        }
    }
    if (t == 0) break;
}
```

- **Ciclo de ordenação**;
- Inicializa-se o **teste de troca**;
- Vai percorrer-se o vector para testar eventuais trocas;
- Se $r1[i1] > r1[i1 + 1]$ vão **trocar-se** os seus valores;
- Para trocar dois valores, usamos uma **variável intermédia**, depois trocam-se os valores;
- Havendo troca marca-se '**t=1**';
- Se houve trocas (**t==1**) **repete-se**; caso contrário **sai-se** do loop.

Ordenação (método de bubble sort) - (Prog08_05.c)

Seja um vector **r1** que se pretende ordenar por **ordem crescente**

```
while (1)
{
    t = 0;
    for (i1=0 ; i1<qt - 1 ;++i1)
    {
        if (r1[i1] > r1[i1+1])
        {
            r0 = r1[i1];
            r1[i1] = r1[i1+1];
            r1[i1+1] = r0;
            t = 1;
        }
    }
    if (t == 0) break;
}
```

- **Ciclo de ordenação**;
- Inicializa-se o **teste de troca**;
- Vai percorrer-se o vector para testar eventuais trocas;
- Se $r1[i1] > r1[i1 + 1]$ vão **trocar-se** os seus valores;
- Para trocar dois valores, usamos uma **variável intermédia**, depois trocam-se os valores;
- Havendo troca marca-se '**t=1**';
- Se houve trocas (**t==1**) **repete-se**; caso contrário **sai-se** do loop.

Ver: https://en.wikipedia.org/wiki/Bubble_sort

Ordenação - Exemplo 2 (Prog08_06.c)

- Sendo a **ordenação** uma **tarefa bem definida**, podemos criar uma **função** onde ela será realizada ('Prog08_06.c').

Ordenação - Exemplo 2 (Prog08_06.c)

- Sendo a **ordenação** uma **tarefa bem definida**, podemos criar uma **função** onde ela será realizada ('Prog08_06.c').
- O código é idêntico ao anterior, no entanto, vejamos como se escreve o **cabeçalho da função** e como ela **é chamada**.

Ordenação - Exemplo 2 (Prog08_06.c)

- Sendo a **ordenação** uma **tarefa bem definida**, podemos criar uma **função** onde ela será realizada ('Prog08_06.c').
- O código é idêntico ao anterior, no entanto, vejamos como se escreve o **cabeçalho da função** e como ela **é chamada**.
- A função de ordenação **bubble_sort** irá receber dois argumentos **r1** (vector dos números a ordenar) e **qt** (quantos são):

Ordenação - Exemplo 2 (Prog08_06.c)

- Sendo a **ordenação** uma **tarefa bem definida**, podemos criar uma **função** onde ela será realizada ('Prog08_06.c').
- O código é idêntico ao anterior, no entanto, vejamos como se escreve o **cabeçalho da função** e como ela **é chamada**.
- A função de ordenação **bubble_sort** irá receber dois argumentos **r1** (vector dos números a ordenar) e **qt** (quantos são):

```
void bubble_sort (int *r1, int qt) { ... }
```


Ordenação - Exemplo 2 (Prog08_06.c)

- Sendo a **ordenação** uma **tarefa bem definida**, podemos criar uma **função** onde ela será realizada ('Prog08_06.c').
- O código é idêntico ao anterior, no entanto, vejamos como se escreve o **cabeçalho da função** e como ela **é chamada**.
- A função de ordenação **bubble_sort** irá receber dois argumentos **r1** (vector dos números a ordenar) e **qt** (quantos são):

```
void bubble_sort (int *r1, int qt) { ... }
```

em que tipo **void** significa que **não é retornado qualquer valor**.

Ordenação - Exemplo 2 (Prog08_06.c)

- Sendo a **ordenação** uma **tarefa bem definida**, podemos criar uma **função** onde ela será realizada ('Prog08_06.c').
- O código é idêntico ao anterior, no entanto, vejamos como se escreve o **cabeçalho da função** e como ela **é chamada**.
- A função de ordenação **bubble_sort** irá receber dois argumentos **r1** (vector dos números a ordenar) e **qt** (quantos são):

```
void bubble_sort (int *r1, int qt) { ... }
```

em que tipo **void** significa que **não é retornado qualquer valor**.

- Como vimos, quando temos uma **variável dimensionada**, o **nome** é o **ponteiro para o início da posição de memória** em que ela se encontra, assim, '**int** ***r1**', quer dizer que '**r1**' é um **ponteiro para inteiros** e não um inteiro.

Ordenação - Exemplo 2 (Prog08_06.c)

- Sendo a **ordenação** uma **tarefa bem definida**, podemos criar uma **função** onde ela será realizada ('Prog08_06.c').
- O código é idêntico ao anterior, no entanto, vejamos como se escreve o **cabeçalho da função** e como ela **é chamada**.
- A função de ordenação **bubble_sort** irá receber dois argumentos **r1** (vector dos números a ordenar) e **qt** (quantos são):

```
void bubble_sort (int *r1, int qt) { ... }
```

em que tipo **void** significa que **não é retornado qualquer valor**.

- Como vimos, quando temos uma **variável dimensionada**, o **nome** é o **ponteiro para o início da posição de memória** em que ela se encontra, assim, '**int** ***r1**', quer dizer que '**r1**' é um **ponteiro para inteiros** e não um inteiro.
- Ou seja, se '**r1**' é um ponteiro para um '**int**', então, '***r1**' é o valor do '**int**' que se encontra na **posição** '**r1**'.

Ordenação - Exemplo 2 (Prog08_06.c)

- Sendo a **ordenação** uma **tarefa bem definida**, podemos criar uma **função** onde ela será realizada ('Prog08_06.c').
- O código é idêntico ao anterior, no entanto, vejamos como se escreve o **cabeçalho da função** e como ela **é chamada**.
- A função de ordenação **bubble_sort** irá receber dois argumentos **r1** (vector dos números a ordenar) e **qt** (quantos são):

```
void bubble_sort (int *r1, int qt) { ... }
```

em que tipo **void** significa que **não é retornado qualquer valor**.

- Como vimos, quando temos uma **variável dimensionada**, o **nome** é o **ponteiro para o início da posição de memória** em que ela se encontra, assim, '**int** ***r1**', quer dizer que '**r1**' é um **ponteiro para inteiros** e não um inteiro.
- Ou seja, se '**r1**' é um ponteiro para um '**int**', então, '***r1**' é o valor do '**int**' que se encontra na **posição 'r1'**.
- Para chamarmos a função: **bubble_sort** (**r1**, **qt**);

Ordenação - Exemplo 2 (Prog08_07.c)

- Aproveitemos o que vimos sobre **valores** e **ponteiros** para construir a função **troca_valores** que terá por tarefa **trocar dois números**.

Ordenação - Exemplo 2 (Prog08_07.c)

- Aproveitemos o que vimos sobre **valores** e **ponteiros** para construir a função **troca_valores** que terá por tarefa **trocar dois números**.
- Como vimos, se queremos que as **alterações nos valores** das variáveis **permaneçam após a execução** de uma função, devemos efectuá-las nas respectivas **posições de memória**.

Ordenação - Exemplo 2 (Prog08_07.c)

- Aproveitemos o que vimos sobre **valores** e **ponteiros** para construir a função **troca_valores** que terá por tarefa **trocar dois números**.
- Como vimos, se queremos que as **alterações nos valores** das variáveis **permaneçam após a execução** de uma função, devemos efectuá-las nas respectivas **posições de memória**.
- Temos **dois modos** equivalentes para o dizer à **função**:

Ordenação - Exemplo 2 (Prog08_07.c)

- Aproveitemos o que vimos sobre **valores** e **ponteiros** para construir a função **troca_valores** que terá por tarefa **trocar dois números**.
- Como vimos, se queremos que as **alterações nos valores** das variáveis **permaneçam após a execução** de uma função, devemos efectuá-las nas respectivas **posições de memória**.
- Temos **dois modos** equivalentes para o dizer à **função**:
 - 1 Através do **ponteiro para** ('&'), ou seja se **r1[i1]** é o **valor do i1^{esimo} elemento**, o ponteiro para ele será **'&r1[i1]'**:

Ordenação - Exemplo 2 (Prog08_07.c)

- Aproveitemos o que vimos sobre **valores** e **ponteiros** para construir a função **troca_valores** que terá por tarefa **trocar dois números**.
- Como vimos, se queremos que as **alterações nos valores** das variáveis **permaneçam após a execução** de uma função, devemos efectuá-las nas respectivas **posições de memória**.
- Temos **dois modos** equivalentes para o dizer à **função**:
 - 1 Através do **ponteiro para** ('&'), ou seja se **r1[i1]** é o **valor do i1^{esimo} elemento**, o ponteiro para ele será '**&r1[i1]**':
troca_valores (&r1[i1], &r1[i1 + 1]);

Ordenação - Exemplo 2 (Prog08_07.c)

- Aproveitemos o que vimos sobre **valores** e **ponteiros** para construir a função **troca_valores** que terá por tarefa **trocar dois números**.
- Como vimos, se queremos que as **alterações nos valores** das variáveis **permaneçam após a execução** de uma função, devemos efectuá-las nas respectivas **posições de memória**.
- Temos **dois modos** equivalentes para o dizer à **função**:
 - 1 Através do **ponteiro para** ('&'), ou seja se **r1[i1]** é o **valor do i1^{esimo} elemento**, o ponteiro para ele será '**&r1[i1]**':
troca_valores (&r1[i1], &r1[i1 + 1]);
 - 2 Se **r1** é o ponteiro para o primeiro elemento (o '**0**' não esquecer), então o ponteiro para o segundo será **r1+1** e, em geral, o ponteiro para o elemento **k** será **r1+k²**

²**Nota:** Nesta operação, o compilador de **C** verifica o espaço ocupado por um elemento e move-se por saltos desse tamanho.

Ordenação - Exemplo 2 (Prog08_07.c)

- Aproveitemos o que vimos sobre **valores** e **ponteiros** para construir a função **troca_valores** que terá por tarefa **trocar dois números**.
- Como vimos, se queremos que as **alterações nos valores** das variáveis **permaneçam após a execução** de uma função, devemos efectuá-las nas respectivas **posições de memória**.
- Temos **dois modos** equivalentes para o dizer à **função**:
 - 1 Através do **ponteiro para** ('&'), ou seja se **r1[i1]** é o **valor do i1^{esimo} elemento**, o ponteiro para ele será '**&r1[i1]**':
troca_valores (&r1[i1], &r1[i1 + 1]);
 - 2 Se **r1** é o ponteiro para o primeiro elemento (o '0' não esquecer), então o ponteiro para o segundo será **r1+1** e, em geral, o ponteiro para o elemento **k** será **r1+k²**
troca_valores (r1+i1, r1+(i1+1));

²**Nota:** Nesta operação, o compilador de **C** verifica o espaço ocupado por um elemento e move-se por saltos desse tamanho.

Ordenação - Exemplo 2 (Prog08_07.c)

Vejamos como construir a função **troca_valores**:

Ordenação - Exemplo 2 (Prog08_07.c)

Vejamos como construir a função **troca_valores**:

- Ela não vai **retornar valores**, logo é do tipo **void**;

Ordenação - Exemplo 2 (Prog08_07.c)

Vejamos como construir a função **troca_valores**:

- Ela não vai **retornar valores**, logo é do tipo **void**;
- Por outro lado recebe **dois ponteiros para inteiros**, logo, as suas declarações são do tipo **'int *n1'**;

Ordenação - Exemplo 2 (Prog08_07.c)

Vejam como construir a função **troca_valores**:

- Ela não vai **retornar valores**, logo é do tipo **void**;
- Por outro lado recebe **dois ponteiros para inteiros**, logo, as suas declarações são do tipo '**int *n1**';
- Se **n1** é o **ponteiro** para a memória, então ***n1**³ será o seu **valor**, isto é, o '**número desejado**'.

³Não esquecer que '***ponteiro**' é o valor para o qual o **ponteiro** aponta

Ordenação - Exemplo 2 (Prog08_07.c)

Vejam como construir a função **troca_valores**:

- Ela não vai **retornar valores**, logo é do tipo **void**;
- Por outro lado recebe **dois ponteiros para inteiros**, logo, as suas declarações são do tipo '**int *n1**';
- Se **n1** é o **ponteiro** para a memória, então ***n1**³ será o seu **valor**, isto é, o '**número desejado**'.
- O código da função **troca_valores** será então:

```
void troca_valores (int *n1, int *n2)
{
    int i1 ;
    i1 = (*n1);
    (*n1) = (*n2);
    (*n2) = i1;
}
```

³Não esquecer que '***ponteiro**' é o valor para o qual o **ponteiro** aponta

Ordenação - Exemplo 3 (Prog05_12.c)

- Podemos utilizar os conhecimentos que adquirimos de ordenação de números para os aplicar à **função logística**;

Ordenação - Exemplo 3 (Prog05_12.c)

- Podemos utilizar os conhecimentos que adquirimos de ordenação de números para os aplicar à **função logística**;
- No programa '**Prog05_11.c**' tínhamos calculado as órbitas periódicas num certo intervalo do parâmetro $r \in [r1, r2]$ e guardado os seus valores num vector;

Ordenação - Exemplo 3 (Prog05_12.c)

- Podemos utilizar os conhecimentos que adquirimos de ordenação de números para os aplicar à **função logística**;
- No programa '**Prog05_11.c**' tínhamos calculado as órbitas periódicas num certo intervalo do parâmetro $r \in [r1, r2]$ e guardado os seus valores num vector;
- Agora, no programa '**Prog05_12.c**', depois de calcularmos os **valores** para cada órbita periódica vamos **ordená-los**.

Ordenação - Exemplo 3 (Prog05_12.c)

- Podemos utilizar os conhecimentos que adquirimos de ordenação de números para os aplicar à **função logística**;
- No programa '**Prog05_11.c**' tínhamos calculado as órbitas periódicas num certo intervalo do parâmetro $r \in [r1, r2]$ e guardado os seus valores num vector;
- Agora, no programa '**Prog05_12.c**', depois de calcularmos os **valores** para cada órbita periódica vamos **ordená-los**.
- Para tal, copiamos as **funções** que criámos no programa '**Prog08_07.c**' e alteramo-las para **ordenarem** reais em **dupla precisão (double)**.

Ordenação - Exemplo 4 (Prog08_08.c)

- Como último exemplo, podemos ver o programa '**Prog08_08.c**' idêntico a '**Prog08_07.c**' em que se substituiu o método '**bubble sort**' por '**quick sort**'.

Ordenação - Exemplo 4 (Prog08_08.c)

- Como último exemplo, podemos ver o programa '**Prog08_08.c**' idêntico a '**Prog08_07.c**' em que se substituiu o método '**bubble sort**' por '**quick sort**'.
- A comparação dos tempos de cálculo dos dois métodos é bem evidente se usarmos mais de 100 000 números aleatórios.

Ordenação - Exemplo 4 (Prog08_08.c)

- Como último exemplo, podemos ver o programa '**Prog08_08.c**' idêntico a '**Prog08_07.c**' em que se substituiu o método '**bubble sort**' por '**quick sort**'.
- A comparação dos tempos de cálculo dos dois métodos é bem evidente se usarmos mais de 100 000 números aleatórios.
- A biblioteca de **C** dispõe de uma função de ordenação que usa o método '**quick sort**'. Essa função designa-se por '**qsort**'.

Ordenação - Exemplo 4 (Prog08_08.c)

- Como último exemplo, podemos ver o programa '**Prog08_08.c**' idêntico a '**Prog08_07.c**' em que se substituiu o método '**bubble sort**' por '**quick sort**'.
- A comparação dos tempos de cálculo dos dois métodos é bem evidente se usarmos mais de 100 000 números aleatórios.
- A biblioteca de **C** dispõe de uma função de ordenação que usa o método '**quick sort**'. Essa função designa-se por '**qsort**'.
- Podem encontrar-se exemplificações animadas em:

Ordenação - Exemplo 4 (Prog08_08.c)

- Como último exemplo, podemos ver o programa '**Prog08_08.c**' idêntico a '**Prog08_07.c**' em que se substituiu o método '**bubble sort**' por '**quick sort**'.
- A comparação dos tempos de cálculo dos dois métodos é bem evidente se usarmos mais de 100 000 números aleatórios.
- A biblioteca de **C** dispõe de uma função de ordenação que usa o método '**quick sort**'. Essa função designa-se por '**qsort**'.
- Podem encontrar-se exemplificações animadas em:
 - http://en.wikipedia.org/wiki/Bubble_sort
 - http://en.wikipedia.org/wiki/Quick_sort

Nota sobre operadores referência e dereferência

- **Operador de referência** ('&'): quando aplicado a uma **variável** fornece o seu **endereço** na memória;

Nota sobre operadores referência e dereferência

- **Operador de referência** ('&'): quando aplicado a uma **variável** fornece o seu **endereço** na memória;
- **Operador de dereferência** ('*'): quando aplicado a um **ponteiro** retorna o **valor** contido na sua posição de memória para a qual aponta;

Nota sobre operadores referência e dereferência

- **Operador de referência** ('&'): quando aplicado a uma **variável** fornece o seu **endereço** na memória;
- **Operador de dereferência** ('*'): quando aplicado a um **ponteiro** retorna o **valor** contido na sua posição de memória para a qual aponta;
- A **declaração dum ponteiro** é feita, **sempre**, à custa do **operador de dereferência**, ou seja, um **ponteiro** é definido à custa do **valor para o qual aponta**.

Nota sobre operadores referência e dereferência

- **Operador de referência** ('&'): quando aplicado a uma **variável** fornece o seu **endereço** na memória;
- **Operador de dereferência** ('*'): quando aplicado a um **ponteiro** retorna o **valor** contido na sua posição de memória para a qual aponta;
- A **declaração dum ponteiro** é feita, **sempre**, à custa do **operador de dereferência**, ou seja, um **ponteiro** é definido à custa do **valor para o qual aponta**.
- Exemplo:

<code>int x;</code>	'x' é uma variável do tipo 'int'	<code>x = 4;</code>
<code>int *p;</code>	'p' é um ponteiro para um 'int'	<code>p = &x;</code>

Assim, o valor para o qual 'p' aponta ('*p') é '4'.