

# Busca e Ordenação

Ordenação em C/C++

---

Prof. Edson Alves - UnB/FGA

2020

1. Ordenação em C
2. Ordenação em C++

# Ordenação em C

---

# Função qsort()

- A biblioteca `stdlib.h` da linguagem C contém a função `qsort()`, a qual implementa o algoritmo *quicksort*
- A assinatura da função `qsort()` é

```
void qsort(void *base, size_t nmemb, size_t size,  
          int (*compar)(const void *, const void *));
```
- O parâmetro `base` é o ponteiro para o primeiro elemento do vetor a ser ordenado
- Como o *quicksort* é um algoritmo de ordenação *in-place*, o vetor apontado por `base` será modificado pela função `qsort()`
- O parâmetro `nmemb` deve indicar o número de elementos a serem ordenados
- O parâmetro `size` indica o tamanho de um elemento, em *bytes*

# Função de comparação

- O último parâmetro da função `qsort` é um ponteiro para a função de comparação `compar`
- Esta função deve receber dois ponteiros constantes `a` e `b` do tipo `void *`
- O retorno deve ser um número inteiro que representa a relação entre os ponteiros:
  1. zero, se `a` e `b` são iguais
  2. negativo, se `a` é menor do que `b`
  3. positivo, se `a` é maior do que `b`
- Como os parâmetros são ponteiros do tipo `void *`, é preciso fazer a coerção dos mesmos para o tipo apropriado na implementação

## Exemplo de uso da função qsort()

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 // Pares em ordem crescente, seguidos de ímpares em ordem decrescente
5 int compare(const void *a, const void *b)
6 {
7     int x = *((int *) a);
8     int y = *((int *) b);
9
10    int rx = x % 2, ry = y % 2;
11
12    if (rx == ry)
13        return rx ? y - x : x - y;
14    else
15        return rx ? 1 : -1;
16 }
17
```

## Exemplo de uso da função qsort()

```
18 int main()
19 {
20     int ns[] { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 }, N = 10;
21
22     qsort(ns, N, sizeof(int), compare);
23
24     for (int i = 0; i < N; ++i)
25         printf("%d%c", ns[i], " \n"[i + 1 == N]);
26
27     return 0;
28 }
```

# Ordenação em C++

---



# Algoritmos de ordenação em C++

- A biblioteca `algorithm` da linguagem C++ oferece 4 algoritmos de ordenação: `sort()`, `stable_sort()`, `partial_sort()` e `nth_element()`
- As interfaces destas funções são semelhantes, e a diferença entre elas está no comportamento de cada uma
- Se indicada, a comparação entre dois elementos a serem ordenados é feita por uma função de comparação binária  $f(a, b)$ , que deve retornar verdadeiro se  $a$  precede  $b$  na ordenação, e falso, em caso contrário
- Se a função de comparação for omitida, a comparação será feita através do operador `<`
- A complexidade média dos três primeiros algoritmos é  $O(N \log N)$ , e a função `nth_element` tem complexidade média  $O(N)$

# Função sort()

- A função `sort()` implementa um algoritmo de ordenação instável, *in-place*, com complexidade média  $O(N \log N)$
- Duas assinaturas possíveis da função `sort()` são

```
template<class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```
- Ao final da execução, o intervalo `[first, last)` estará ordenado, de acordo com o operador `<` ou o comparador `comp`, respectivamente
- A segunda assinatura permite uma maior flexibilidade, pois permite a customização do critério de comparação
- Se o critério é apenas a substituição do operador `<` pelo operador `>`, basta usar a estrutura `greater` da biblioteca padrão do C++

## Exemplo de uso da função sort()

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 int main() {
6     vector<int> ns { 8, 5, 1, 3, 2, 9, 6, 4, 10, 7 };
7
8     sort(ns.begin(), ns.end());
9
10    for (size_t i = 0; i < ns.size(); ++i)
11        cout << ns[i] << (i + 1 == ns.size() ? "\n" : ", ");
12
13    sort(ns.begin(), ns.end(), greater<int>());
14
15    for (size_t i = 0; i < ns.size(); ++i)
16        cout << ns[i] << (i + 1 == ns.size() ? "\n" : ", ");
17
18    return 0;
19 }
```

# Comparadores

- O comparador pode ser implementado de 3 maneiras: como uma função binária, como uma função binária anônima (*lambda*) ou como uma estrutura com o operador booleano binário (`()`)
- A primeira forma tem como vantagem a familiaridade da declaração de funções, mas amplia o código com uma função de uso específico
- A terceira forma traz o mesmo problema, mas o uso da estrutura reduz a poluição do espaço de nomes
- A segunda forma evita os problemas de nomes por usar uma função anônima, e a declaração na chamada do algoritmo de ordenação reduz a distância entre declaração e uso
- Contudo, tem sintaxe menos intuitiva e pode levar à duplicação de código, caso precise ser utilizada mais de uma vez
- Alternativamente, pode-se implementar o operador `<` na própria classe dos objetos a serem comparados e usar a primeira assinatura

## Exemplo de uso de comparadores

```
1 #include <iostream>
2 #include <algorithm>
3
4 using namespace std;
5
6 void print(const string as[], int N) {
7     for (int i = 0; i < N; ++i)
8         cout << as[i] << (i + 1 == N ? "\n" : ", ");
9 }
10
11 // Primeira forma: não diferencia maiúsculas de minúsculas
12 bool compare(const string& a, const string& b)
13 {
14     string x, y;
15     auto to_lower = [](char c) { return tolower(c); };
16
17     transform(a.begin(), a.end(), back_inserter(x), to_lower);
18     transform(b.begin(), b.end(), back_inserter(y), to_lower);
19
20     return x < y;
21 }
```

## Exemplo de uso de comparadores

```
22
23 // Terceira forma: primeiro por tamanho, depois lexicográfico
24 struct Compare
25 {
26     bool operator()(const string& a, const string& b)
27     {
28         int N = a.size(), M = b.size();
29         return N != M ? N < M : a < b;
30     }
31 };
32
33 int main()
34 {
35     string as[] { "verde", "amarelo", "Vermelho", "Branco", "Preto",
36                 "azul" };
37     int N = 6;
38
39     // Ordenação lexicográfica
40     sort(as, as + N);
41     print(as, N);           // Branco, Preto, Vermelho, amarelo, azul, verde
42
```

## Exemplo de uso de comparadores

```
43 // Ordenação lexicográfica inversa
44 sort(as, as + N, greater<string>());
45 print(as, N);          // verde, azul, amarelo, Vermelho, Preto, Branco
46
47 // Ordenação case-insensitive
48 sort(as, as + N, compare);
49 print(as, N);          // amarelo, azul, Branco, Preto, verde, Vermelho
50
51 // Ordenação por tamanho, depois lexicográfica
52 sort(as, as + N, Compare());
53 print(as, N);          // azul, Preto, verde, Branco, amarelo, Vermelho
54
55 // Segunda forma
56 // Primeiro inicial minúscula, depois maiúscula, lexicográfica
57 sort(as, as + N, [](const string& a, const string& b) {
58     auto x = islower(a[0]), y = islower(b[0]);
59     return x == y ? a < b : (x ? true : false); });
60 print(as, N);          // amarelo, azul, verde, Branco, Preto, Vermelho
61
62 return 0;
63 }
```

## Função `stable_sort()`

- A função `stable_sort()` implementa um algoritmo de ordenação estável, *in-place*, com complexidade média  $O(N \log N)$
- Duas assinaturas possíveis da função `stable_sort()` são

```
template<class RandomAccessIterator>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);
```
- Ao contrário da função `sort()`, a função `stable_sort()` preserva a ordem relativa dos elementos considerados iguais
- É possível transformar qualquer algoritmo de ordenação instável em um algoritmo estável, adicionado-se a cada elemento da sequência um identificador inteiro único de posição
- Contudo, esta adaptação tem custo de memória  $O(N)$ , além de implicar ou na alteração dos elementos da sequência ou no uso de pares ao invés dos elementos



## Exemplo de uso da função `stable_sort()`

```
1 // Baseado no exemplo ilustrado no cppman
2 #include <bits/stdc++.h>
3
4 using namespace std;
5
6 int main()
7 {
8     vector<double> xs { 2.7, 2.2, 1.8, 1.3, 1.1, 3.2, 2.9 }, ys = xs;
9     auto cmp = [](double a, double b) { return int(a) < int(b); };
10
11     sort(xs.begin(), xs.end());           // 1.1 1.3 1.8 2.2 2.7 2.9 3.2
12     stable_sort(ys.begin(), ys.end(), cmp); // 1.8 1.3 1.1 2.7 2.2 2.9 3.2
13
14     return 0;
15 }
```

## Função `partial_sort()`

- A função `partial_sort()` tem assinaturas ligeiramente diferentes das duas funções citadas anteriormente, por contar com um terceiro parâmetro:

```
template <class RandomAccessIterator>
void partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
                  RandomAccessIterator last);
```

```
template <class RandomAccessIterator, class Compare>
void partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
                  RandomAccessIterator last, Compare comp);
```

- Ela é semelhante à rotina de pivoteamento do *quicksort*, ordenando os elementos que ficam à esquerda da posição indicada e deixando os demais elementos em uma ordem não especificada
- Em outras palavras, ela ordena apenas os elementos menores ou iguais ao elemento apontado por `middle`

## Exemplo de uso da função `partial_sort()`

```
1 #include <iostream>
2 #include <vector>
3 #include <algorithm>
4
5 using namespace std;
6 using score = pair<int, string>;
7
8 void topN(vector<score>& scores, size_t N)
9 {
10     partial_sort(scores.begin(), scores.begin() + N, scores.end(),
11                 [](const score& a, const score& b) {
12                     return a.first == b.first ? a.second < b.second :
13                         a.first > b.first;
14                 });
15
16     for (size_t i = 0; i < min(N, scores.size()); ++i)
17         cout << i + 1 << ". " << scores[i].second << " - "
18             << scores[i].first << " pontos\n";
19 }
20
```

## Exemplo de uso da função `partial_sort()`

```
21 int main()
22 {
23     vector<score> scores { score(50, "Alonso"), score(207, "Bottas"),
24                          score(331, "Hamilton"), score(53, "Hulkenberg"),
25                          score(53, "Perez"), score(53, "Magnussen"),
26                          score(196, "Raikkonen"), score(146, "Ricardo"),
27                          score(173, "Verstappen"), score(264, "Vettel") };
28
29     topN(scores, 5);
30
31     for (const auto& [s, name] : scores)
32         cout << name << " - " << s << " pontos\n";
33
34     return 0;
35 }
```

## Função `nth_element()`

- Uma função semelhante à `partial_sort()` é a função `nth_element()`, que posiciona corretamente apenas o elemento que ocuparia a  $n$ -ésima posição do vetor, caso estivesse ordenado
- A complexidade média é  $O(N)$
- Duas assinaturas possíveis são

```
template <class RandomAccessIterator>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                 RandomAccessIterator last);
```

```
template <class RandomAccessIterator, class Compare>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                 RandomAccessIterator last, Compare comp);
```

## Exemplo de uso da função nth\_element()

```
1 #include <iostream>
2 #include <algorithm>
3
4 using namespace std;
5
6 int median(int xs[], int N)
7 {
8     nth_element(xs, xs + N/2, xs + N);
9
10    return xs[N/2];
11 }
12
13 int main()
14 {
15     int xs[] { 7, 1, 4, 8, 2, 6, 5, 9, 3 }, N = 9;
16
17     cout << median(xs, N) << endl;
18
19     return 0;
20 }
```

1. **KERNIGHAN**, Bryan; **RITCHIE**, Dennis. *The C Programming Language*, 1978.
2. **STROUSTROUP**, Bjarne. *The C++ Programming Language*, 2013.
3. Páginas manuais do Linux<sup>1</sup>
4. Cppman<sup>2</sup>

---

<sup>1</sup>Comando man do Linux

<sup>2</sup><https://github.com/aitjcize/cppman>