



UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE LINGUAGENS E SISTEMAS DE COMPUTAÇÃO
DISCIPLINA ELC 118 – PESQUISA E ORDENAÇÃO DE DADOS

Algoritmos de Ordenação em C++

PROF.º HENRIQUE MICHEL PERSCH
Bacharel em Sistemas de Informação – UFSM

Contato: hpersch@inf.ufsm.br

Introdução

- Algoritmo de ordenação em ciência da computação é um algoritmo que coloca os elementos de uma dada sequência em uma certa ordem - em outras palavras, efetua sua ordenação completa ou parcial.
- As linguagens de programação já possuem métodos de ordenação.
- Problema: encontrar um número de telefone em uma lista telefônica ou busca de um livro em uma biblioteca.

Objetivos

- Corresponde ao processo de rearranjar um conjunto de objetos em uma ordem ascendente ou descendente.
- Facilitar a recuperação posterior de itens do conjunto ordenado.
- São largamente utilizados em Sistemas Gerenciadores de Banco de Dados (SGBDs).
- Em alguns casos é necessário fazer a avaliação da ordenação no melhor, médio e pior caso.
- Objetivo da Aula: Demonstrar as funções de ordenação existentes em C++ e outras possíveis funções.

Ordenação em C++

- Funções que já implementam algoritmos de Ordenação em C++:
 - ✓ Sort
 - ✓ Stable_sort
 - ✓ Partial_sort

Sort

- Apresenta duas possibilidades de Ordenação:
 - `std :: sort (myvector.begin (), myvector.end)`
 - `std :: sort (myvector.begin (), myvector.end (), myCompFuncion)`
- Para utilizar esse algoritmo, precisamos adicionar a biblioteca “`#include <algorithm>`”
- Parâmetros utilizados:
 - `myvector.begin()`: Ponteiro para o primeiro elemento do vetor que se deseja ordenar.
 - `myvector.end()`: Ponteiro para o último elemento do vetor que se deseja ordenar.
 - `myCompFuncion()` - Opcional: Passa uma função que irá retornar `TRUE` ou `FALSE` para ordenar os elementos no caso de utilização de estruturas de dados criadas pelo programador.
- Essa função não retorna nada, pois sua ordenação é feita através de Ponteiros.

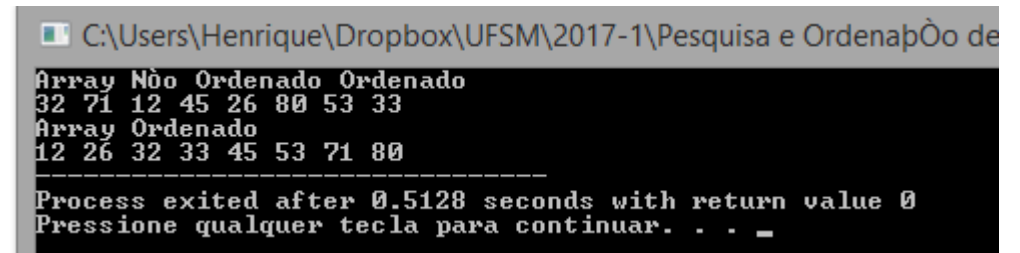
Sort – Exemplo 1

- Quando se trata de um tipo conhecido (int, double, string) podemos ordenar utilizando o operador “>”, nesse caso, podemos utilizar a forma simples do *sort*, conforme o exemplo abaixo:

```
#include <iostream>
#include <algorithm>

using namespace std;

int main()
{
    int myints[] = {32,71,12,45,26,80,53,33};
    std::vector<int> myvector (myints, myints+8);
    //Mostra como o vetor está.
    cout << "Array Não Ordenado Ordenado" << endl;
    for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
        cout << *it << " ";
    //Chama a função de ordenação
    sort(myvector.begin(), myvector.end());
    cout << "\nArray Ordenado" << endl;
    for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
        cout << *it << " ";
    return 0;
}
```



```
C:\Users\Henrique\Dropbox\UFSM\2017-1\Pesquisa e OrdenapÕo de
Array Não Ordenado Ordenado
32 71 12 45 26 80 53 33
Array Ordenado
12 26 32 33 45 53 71 80
-----
Process exited after 0.5128 seconds with return value 0
Pressione qualquer tecla para continuar. . . _
```

Sort – Exemplo 2

- Se quisermos ordenar em outra ordem ou ordenar um vetor de structs (não é trivial comparar), precisamos criar uma função de comparação.

```
using namespace std;

bool compare (int i, int j)
{
    return (i<j);
}

int main()
{
    int myints[] = {32,71,12,45,26,80,53,33};
    std::vector<int> myvector (myints, myints+8);
    //Mostra como o vetor está.
    cout << "Array Não Ordenado Ordenado" << endl;
    for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
        cout << *it << " ";
    //Chama a função de ordenação
    sort(myvector.begin(), myvector.end(), compare);
    cout << "\nArray Ordenado" << endl;
    for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
        cout << *it << " ";
    return 0;
}
```

```
Array Não Ordenado Ordenado
32 71 12 45 26 80 53 33
Array Ordenado
12 26 32 33 45 53 71 80
-----
Process exited after 0.6735 seconds with return value 0
Pressione qualquer tecla para continuar. . . _
```

Sort – Exemplo 3

- Ordenação utilizando um tipo de dado, criado pelo programador.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>

using namespace std;

struct Pessoa
{
    string nome;
    int idade;
    string cor;
};

bool sortNome(const Pessoa &lhs, const Pessoa &rhs) { return lhs.nome < rhs.nome; }

bool sortIdade(const Pessoa &lhs, const Pessoa &rhs) { return lhs.idade < rhs.idade; }

bool sortCor(const Pessoa &lhs, const Pessoa &rhs) { return lhs.cor < rhs.cor; }
```



```

int main()
{
    vector<Pessoa> pessoa(3);
    pessoa[0].nome = "Joao"; pessoa[0].idade = 20; pessoa[0].cor = "Azul";
    pessoa[1].nome = "Pedro"; pessoa[1].idade = 15; pessoa[1].cor = "Vermelho";
    pessoa[2].nome = "Maria"; pessoa[2].idade = 55; pessoa[2].cor = "Laranja";

    cout << "Ordenacao por Nome: \n";
    sort(pessoa.begin(), pessoa.end(), sortNome);
    for (Pessoa &n : pessoa)
        cout << n.nome << " ";
    cout << endl;

    cout << "Ordenacao por Idade: \n";
    sort(pessoa.begin(), pessoa.end(), sortIdade);
    for (Pessoa &n : pessoa)
        cout << n.idade << " ";
    cout << endl;

    cout << "Ordenacao por Cor: \n";
    sort(pessoa.begin(), pessoa.end(), sortCor);
    for (Pessoa &n : pessoa)
        cout << n.cor << " ";
    cout << endl;
}

```

```

Ordenacao por Nome:
Joao Maria Pedro
Ordenacao por Idade:
15 20 55
Ordenacao por Cor:
Azul Laranja Vermelho

-----
Process exited after 0.5193 seconds with return value 0
Pressione qualquer tecla para continuar. . .

```

Stable_Sort

- Apresenta duas possibilidades de Ordenação:
 - `std :: stable_sort (myvector.begin (), myvector.end)`
 - `std :: stable_sort (myvector.begin (), myvector.end (), myCompFuncion)`
- Diferença para o Sort: preserva a ordem relativa dos elementos antes da chamada.
- Parâmetros que são utilizados:
 - `myvector.begin()`: Ponteiro para o primeiro elemento do vetor que se deseja ordenar.
 - `myvector.end()`: Ponteiro para o último elemento do vetor que se deseja ordenar.
 - `myCompFunction()` - Opcional: Passa uma função que irá retornar TRUE ou FALSE para ordenar os elementos no caso de utilização de estruturas de dados criadas pelo programador.
- Essa função não retorna nada, pois sua ordenação é feita através de Ponteiros

Exemplo Stable_Sort

```
using namespace std;
bool compare_as_ints (double i,double j)
{
    return (int(i)<int(j));
}

int main () {
    double mydoubles[] = {3.14, 1.41, 2.72, 4.67, 1.73, 1.32, 1.62, 2.58};

    vector<double> myvector;

    myvector.assign(mydoubles,mydoubles+8);

    cout << "Usando comparação Padrão:";
    stable_sort (myvector.begin(), myvector.end());
    for (std::vector<double>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
        std::cout << ' ' << *it;
    cout << '\n';

    myvector.assign(mydoubles,mydoubles+8);

    cout << "Usando a função:";
    stable_sort (myvector.begin(), myvector.end(), compare_as_ints);
    for (std::vector<double>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
        std::cout << ' ' << *it;
    cout << '\n';

    return 0;
}
```

```
Usando comparação Padrão:
1.32 1.41 1.62 1.73 2.58 2.72 3.14 4.67
Usando a função:
1.41 1.73 1.32 1.62 2.72 2.58 3.14 4.67
-----
Process exited after 0.5725 seconds with return value 0
Pressione qualquer tecla para continuar. . . _
```

Exemplo Stable_Sort – Com Struct

```
using namespace std;
```

```
struct Pessoa {  
    Pessoa(int idade, std::string nome) : idade(idade), nome(nome) { }  
    int idade;  
    std::string nome;  
};
```

```
bool operator<(const Pessoa &lhs, const Pessoa &rhs) {  
    return lhs.idade < rhs.idade;  
}
```

```
bool sortNome(const Pessoa &lhs, const Pessoa &rhs) {  
    return lhs.nome < rhs.nome;  
}
```

```
32, Arthur  
108, Ford  
23, Zaphod  
Pressione qualquer tecla para continuar. . . _
```

```
main()  
{  
    vector<Pessoa> v = {  
        Pessoa(23, "Zaphod"),  
        Pessoa(32, "Arthur"),  
        Pessoa(108, "Ford"),  
    };  
  
    stable_sort(v.begin(), v.end());  
    stable_sort(v.begin(), v.end(), sortNome);  
  
    for (const Pessoa &e : v) {  
        cout << e.idade << ", " << e.nome << "\n";  
    }  
    system("pause");  
}
```

Partial_Sort

- Reorganiza os elementos, de tal forma que os elementos antes do meio ou da quantidade de elementos que deseja ordenar, são os elementos menores de todo o intervalo e são classificados em ordem crescente, enquanto os elementos restantes são deixados sem qualquer ordem específica.
- Apresenta duas possibilidades de Ordenação:
 - `std :: partial_sort (myvector.begin (), myvector.begin ()+numelementos ,myvector.end)`
 - `std :: partial_sort (myvector.begin (), myvector.end()+numelementos , myvector.end, myCompFuncion)`
- Parâmetros que são utilizados:
 - `myvector.begin()`: Ponteiro para o primeiro elemento do vetor que se deseja ordenar.
 - `myvector.begin()+numelementos`: Ponteiro para o meio do vetor que se deseja ordenar.
 - `myvector.end()`: Ponteiro para o último elemento do vetor que se deseja ordenar.
 - `myCompFunction()` - Opcional: Passa uma função que irá retornar TRUE ou FALSE para ordenar os elementos no caso de utilização de estruturas de dados criadas pelo programador.

Exemplo Partial_Sort

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
bool minhaFuncao (int i,int j)
{
    return (i<j);
}

int main () {
    int myints[] = {9,8,7,6,5,4,3,2,1};
    vector<int> meuVector (myints, myints+9);

    // usando o operador padrão <
    partial_sort (meuVector.begin(), meuVector.begin()+6, meuVector.end());

    std::cout << "Vetor Ordenado sem Função até o 6 elemento:";
    for (std::vector<int>::iterator it=meuVector.begin(); it!=meuVector.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << "\n";
    meuVector.assign(myints, myints+9);

    // usando uma função de comparação.
    partial_sort (meuVector.begin(), meuVector.begin()+4, meuVector.end(),minhaFuncao);

    std::cout << "Vetor Ordenado com Função até o 4 elemento:";
    for (std::vector<int>::iterator it=meuVector.begin(); it!=meuVector.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';

    return 0;
}
```

```
Vetor Ordenado sem Função até o 6 elemento: 1 2 3 4 5 6 9 8 7
Vetor Ordenado com Função até o 4 elemento: 1 2 3 4 9 8 7 6 5

-----
Process exited after 0.6334 seconds with return value 0
Pressione qualquer tecla para continuar. . . _
```

Função semelhante:

partial_sort_copy, que realiza uma cópia dos elementos, e ordena-os para dentro de uma outra estrutura.

Funções do C++ para Ordenação

- Além dos algoritmos de ordenação implementados na linguagem do C++ para este fim, podemos também utilizar algumas funções que auxiliam no desenvolvimento de recursos de ordenação em nosso código.
- Temos os seguintes exemplos:
 - ✓ `inplace_merge`
 - ✓ `Heap_Sort`
 - ✓ `Is_sorted`
 - ✓ `Is_permutation`

Inplace_merge

- Essa função, mescla dois intervalos consecutivos ordenados: `[first,middle)` e `[middle,last)`, colocando o resultado no intervalo combinado classificado `[first,last)`.
- Os elementos são comparados usando, o operador `<`, ou uma função de comparação.
- Os elementos em ambas as partes já devem ser ordenados de acordo com estes mesmos critério (`operator<` ou função). O intervalo resultante também é classificado de acordo com isso.
- A função preserva a ordem relativa de elementos com valores equivalentes, com os elementos na primeira faixa que precede os equivalentes na segunda.
- Parâmetros utilizados:
 - `myvector.begin()`: Ponteiro para o primeiro elemento do vetor que se deseja ordenar.
 - `myvector.end()`: Ponteiro para o último elemento do vetor que se deseja ordenar.
 - `myCompFunction()` - Opcional: Passa uma função que irá retornar `TRUE` ou `FALSE` para ordenar os elementos no caso de utilização de estruturas de dados criadas pelo programador.

Exemplo – Inplace_merge

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main () {
    int first[] = {5,10,15,20,25};
    int second[] = {50,40,30,20,10};
    vector<int> v(10);
    vector<int>::iterator it;

    sort (first,first+5);
    sort (second,second+5);

    it=copy (first, first+5, v.begin());
    | copy (second,second+5,it);

    inplace_merge(v.begin(),v.begin()+5,v.end());

    std::cout << "O vetor final fica:\n";
    for (it=v.begin(); it!=v.end(); ++it)
        cout << ' ' << *it;
    cout << '\n';

    return 0;
}
```

```
O vetor final fica:
5 10 10 15 20 20 25 30 40 50
```

```
-----
Process exited after 0.7045 seconds with return value 0
Pressione qualquer tecla para continuar. . .
```

Sort_heap

- Classifica os elementos na faixa de heap em ordem crescente.
- Os elementos são comparados usando operator< para a primeira versão, e comp para o segundo, que deve ser o mesmo usado para construir o heap.
- O intervalo perde suas propriedades como um heap.

6 5 3 1 8 7 2 4

Exemplo Heap_Sort

```
#include <iostream>
#include <algorithm>
#include <vector>

int main () {
    int myints[] = {6,5,3,1,8,7,2,4};
    std::vector<int> v(myints,myints+8);

    std::make_heap(v.begin(),v.end());//cira a árvore binária de elementos
    std::cout << "Valor inicial maximo   : \n" << v.front() << '\n';

    std::pop_heap (v.begin(),v.end()); v.pop_back();
    std::cout << "Pilha máxima apos pop : \n" << v.front() << '\n';

    std::sort_heap (v.begin(),v.end());

    std::cout << "Resultado Final:\n";
    for (unsigned i=0; i<v.size(); i++)
        std::cout << " " << v[i];

    std::cout << '\n';

    return 0;
}
```

```
Valor inicial maximo   :
8
Pilha máxima apos pop :
7
Resultado Final:
1 2 3 4 5 6 7

-----
Process exited after 0.6833 seconds with return value 0
Pressione qualquer tecla para continuar. . .
```

Is_sorted

- Essa função verifica se o intervalo repassado já está ordenado.
- Pode utilizar a verificação padrão, com o operador >, ou pode receber uma função que retorna TRUE ou FALSE para verificar a ordenação.

```
Vetor: 2 3 4 1
Vetor: 2 3 1 4
Vetor: 2 1 4 3
Vetor: 2 1 3 4
Vetor: 1 4 3 2
Vetor: 1 4 2 3
Vetor: 1 3 4 2
Vetor: 1 3 2 4
Vetor: 1 2 4 3
Vetor: 1 2 3 4
0 vetor está Ordenado!
```

```
#include <iostream>
#include <algorithm>
#include <array>

using namespace std;
int main () {
    array<int,4> vetor {2,4,1,3};

    do {
        // realiza as permutações
        prev_permutation(vetor.begin(),vetor.end());

        // imprime cada etapa
        cout << "Vetor:";
        for (int& x:vetor)
            cout << " " << x;
        cout << '\n';

    } while (!is_sorted(vetor.begin(),vetor.end()));

    std::cout << "0 vetor está Ordenado!\n";

    return 0;
}
```

Is_permutation

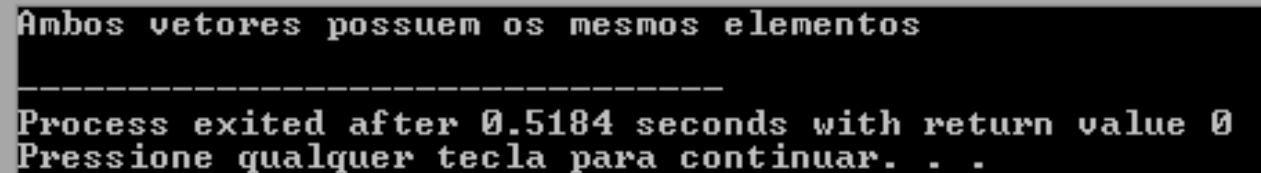
- Compara os elementos de um vetor com aqueles no intervalo que começa no outro vetor e retorna TRUE se todos os elementos em ambos os intervalos são iguais, mesmo em uma ordem diferente.

```
#include <iostream>
#include <algorithm>
#include <array>

using namespace std;
int main () {
    std::array<int,5> vetor1 = {1,2,3,4,5};
    std::array<int,5> vetor2 = {3,1,4,5,2};

    if ( is_permutation (vetor1.begin(), vetor1.end(), vetor2.begin()) )
        cout << "Ambos vetores possuem os mesmos elementos\n";

    return 0;
}
```



Ambos vetores possuem os mesmos elementos

Process exited after 0.5184 seconds with return value 0

Pressione qualquer tecla para continuar. . .

Conclusão

- O uso de funções de ordenação podem facilitar o encontro de alguma informação.
- Diversas funções implementam, em diferentes linguagens, já fazem isso.

Dúvidas!

