Curso de C++ para Resolução de Problemas

Projeto de Ensino e Aprendizagem de Programação para Olimpíadas

Iterator e Sort

- Iterator
- Sort
  - Ordenação Padrão
  - Função de Comparação
  - Ordenação Customizada
  - Ordenação Estável

Iterator

Um **iterator** é qualquer objeto que, apontando para algum elemento em um intervalo de elementos (como um vetor, uma matriz ou um contêiner), tem a capacidade de percorrer os elementos desse intervalo utilizando um conjunto de operadores. Os operadores mais utilizados e garantidos para qualquer tipo de objeto são o operador de incremento **++**, que faz com que o **iterator** acesse a próxima posição do intervalo, e o operador de excluir a referência *****, que acessa o conteúdo do elemento apontado pelo **iterator**.

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

using namespace std;

int main()
    vector<int> v(3);
    for(int i=0; i<(int)v.size(); i++)
        v[i] = (i+1)*10;
    cout << "Os elementos de v são: ";
    vector<int>::iterator it;
    for(it=v.begin(); it!=v.end(); it++)
        cout << *it << " ";
    cout << endl;
    return 0;
}
```

Assim, para declarar um **iterator** utilizamos o padrão `vector<int>::iterator it` tal que é necessário especificar o tipo de contêiner com o qual estamos lidando, neste caso um `vector<int>`, e logo em seguida escrevemos `::iterator`, definindo que a variável nomeada como `it`, neste caso, é um **iterator** do tipo `vector<int>`, assim não pode ser utilizada para qualquer outro tipo de contêiner, como por exemplo `vector<string>` ou `vector<double>`.

No laço de repetição `for` após a declaração do **iterator** temos a estrutura `it=v.begin(); it!=v.end(); it++` indicando que `it` está apontando para o início do contêiner definido, `it=v.begin()`, e vai iterar até chegar ao seu fim, `it!=v.end()`, sendo incrementado de uma em uma posição, `it++`. Dentro do bloco de operações da estrutura de repetição `for` imprimimos o conteúdo apontado por `it` através do uso do operador de excluir a referência.

```
#include <iostream>
#include <vector>
#include <iterator>

using namespace std;

int main(){
    vector<int> v;
    for(int i=0; i<10; i++)
        v.push_back(i+1);
    vector<int>::iterator inicio, it;
    inicio = v.begin();
    it = inicio;
    it++;
    cout << "it está apontando para o elemento " << *it << endl;
    cout << "A distância entre inicio e it é " << distance(inicio,it) <<
endl;
    return 0;
}
```

Para saber a distância, isto é, quantos elementos existem entre dois iteradores, devemos incluir a biblioteca `iterator` e usar a função **distance**, passando como parâmetro os iteradores desejados. Isto pode ser útil quando queremos saber a posição de um elemento em um contêiner, assim sendo possível acessar o seu conteúdo diretamente pelo operador **[]** apenas especificando a distância do **iterator** até o início do contêiner, como mostrado no exemplo acima.

● Sort

A função **sort** é implementada pela biblioteca `algorithm` e declarada no espaço de nomes padrão `namespace std` que temos utilizado até agora. Assim, utilizamos a função **sort** para naturalmente ordenar um conjunto de elementos, seja em ordem crescente, decrescente ou através de uma função de comparação, para casos em que desejamos ordenar um conjunto de elementos em ordem diferenciada ou em que não existe uma ordem definida para o tipo de dados com o qual estamos lidando, por exemplo, uma **struct**.

- Ordenação Padrão

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

using namespace std;

int main(){
    vector<string> v(3);
    v[0] = "banana";
    v[1] = "abacaxi";
    v[2] = "melancia";
    cout << "Conjunto não ordenado: ";
    for(int i=0; i<(int)v.size(); ++i)
        cout << v[i] << " ";
    cout << endl;
    sort(v.begin(),v.end());
    cout << "Conjunto em ordem crescente: ";
    for(int i=0; i<(int)v.size(); ++i)
        cout << v[i] << " ";
    cout << endl;
    sort(v.rbegin(),v.rend());
    cout << "Conjunto em ordem decrescente: ";
    for(int i=0; i<(int)v.size(); ++i)
        cout << v[i] << " ";
    cout << endl;
    return 0;
}
```

Aplicando a função `sort(v.begin(),v.end())` para o contêiner `vector<string> v` obtemos o conjunto em ordem crescente, já que passamos como parâmetro os iteradores para o início e para o fim do contêiner. Já quando aplicamos a função `sort(v.rbegin(),v.rend())` com os iteradores reversos do contêiner como parâmetro obtemos o conjunto em ordem

decrescente. Os iteradores reversos também podem ser utilizados para percorrer um contêiner de forma reversa, ou seja, partindo do último elemento até o primeiro.

- **Função de Comparação**

Uma função de comparação define a maneira com a qual queremos comparar os elementos para estabelecer uma ordem, podendo fazer uso de quaisquer que sejam as suas informações.

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

using namespace std;

struct pessoa{
    string nome;
    int idade;
};

bool compara(pessoa x, pessoa y){
    if(x.nome.size()<y.nome.size())
        return true;
    else
        return false;
}

int main(){
    pessoa p1, p2;
    p1.nome = "Joao", p1.idade = 40;
    p2.nome = "Marta", p2.idade = 7;
    if(compara(p1,p2)){
        cout << p1.nome << " " << p1.idade;
        cout << " vem antes de ";
        cout << p2.nome << " " << p2.idade << endl;
    }
    return 0;
}
```

Então, definimos a função de comparação como do tipo `bool` que retorna se a comparação realizada foi `true` (verdadeira) ou `false` (falsa) com base na ordem estabelecida entre os dois elementos passados como parâmetro para a função `compara`. Neste caso, estamos estabelecendo uma ordem crescente com base somente no tamanho do `nome` da `struct pessoa`. Logo, `"Joao"` vem antes `"Marta"` porque os tamanhos dos nomes são `4` e `5`,

respectivamente. Lembrando que também é possível definir outras funções de comparação com base em quaisquer outras informações não utilizadas da estrutura, neste caso a `idade`, ou em quaisquer outros tipos de dados.

- **Ordenação Customizada**

Mantendo a mesma estrutura `pessoa` e a mesma função `compara` definidas no exemplo anterior podemos aplicar a função **`sort`** para ordenar um conjunto com base no critério estabelecido, seja em ordem crescente ou decrescente, somente acrescentando como parâmetro a função de comparação na chamada da função de ordenação.

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

using namespace std;

struct pessoa{
    string nome;
    int idade;
};

bool compara(pessoa x, pessoa y){
    if(x.nome.size()<y.nome.size())
        return true;
    else
        return false;
}

int main(){
    vector<pessoa> v(3);
    v[0].nome = "Marta", v[0].idade = 7;
    v[1].nome = "Joao", v[1].idade = 40;
    v[2].nome = "Raimundo", v[2].idade = 64;
    cout << "Conjunto não ordenado: ";
    for(int i=0; i<(int)v.size(); ++i)
        cout << v[i].nome << " " << v[i].idade << " ";
    cout << endl;
    sort(v.begin(),v.end(),compara);
    cout << "Conjunto em ordem crescente: ";
   for(int i=0; i<(int)v.size(); ++i)
        cout << v[i].nome << " " << v[i].idade << " ";
```

```
    cout << endl;
    sort(v.rbegin(),v.rend(),compara);
    cout << "Conjunto em ordem decrescente: ";
    for(int i=0; i<(int)v.size(); ++i)
        cout << v[i].nome << " " << v[i].idade << " ";
    cout << endl;
    return 0;
}
```

- **Ordenação Estável**

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

using namespace std;

bool compara_como_inteiros(double x, double y){
    return (int(x)>int(y));
}

int main(){
    double vetor[] = {3.14, 1.41, 2.72, 4.67, 1.73, 1.32, 1.62, 2.58};
    vector<double> v;
    for(int i=0; i<8; i++)
        v.push_back(vetor[i]);
    cout << "Conjunto não ordenado: ";
    for(int i=0; i<(int)v.size(); ++i)
        cout << v[i] << " ";
    cout << endl;
    sort(v.begin(),v.end());
    cout << "Conjunto em ordem padrão crescente: ";
    for(int i=0; i<(int)v.size(); ++i)
        cout << v[i] << " ";
    cout << endl;
    stable_sort(v.begin(),v.end(),compara_como_inteiros);
    cout << "Conjunto em ordem inteira decrescente estável: ";
    for(int i=0; i<(int)v.size(); ++i)
        cout << v[i] << " ";
    cout << endl;
    return 0;
}
```

A única diferença de utilizar a função **`stable_sort`** ao invés da função **`sort`** para ordenar um conjunto de elementos é que a **`stable_sort`** preserva a ordem relativa dos elementos que possuem valores equivalentes, por isso chamamos esse tipo de ordenação de ordenação estável.

PROGRAMATHON

Projeto de Ensino e Aprendizagem de Programação para Olimpíadas

EQUIPE

Reginaldo M. Kuroshu (Coordenador Geral do Projeto)
Alexandre Hild Aono
Bruno Bernardo de Moura
Danilo Gustavo Hansen Laboissiere
Diogo Augusto Hansen Laboissiere
Lucas de Alencar Barbosa
Marcos Castro de Souza
Nathan de Melo Cruz
Rodrigo de Farias Ramires
Thauany Moedano
Victor de Sá Nunes
Willian da Silva Zocolau

APOIO

Pró-Reitoria de Extensão (PROEX-UNIFESP)

REALIZAÇÃO

Instituto de Ciência e Tecnologia (ICT-UNIFESP)

CONTATO

programathon.unifesp@gmail.com

PARA MAIS INFORMAÇÕES ACESSE

programathon-unifesp.github.io