CONTEINERS VECTOR E LIST

Livro Bibl Digital: Conceitos de Computação com o Essencial de C++. HORSTMANN, Cay
Cap 9 Vetores e Arrays – exercícios de programação P9.1 a P9.10
Cap16 Introdução a Estruturas de Dados – exercícios de programação P16.1 a P16.4

Iteradores

- similares a ponteiros, usados para apontar para os elementos dos contêineres.
- armazenam a informação aos tipos específicos de contêineres que eles operam => eles devem ser implementados com o mesmo tipo do contêiner a percorrer.
- contêineres oferecem os métodos **begin()** e **end()** para usar com iteradores.
- operador * é usado para acessar o elemento apontado.

Para criar e usar um iterador deve-se fazer:

```
vector<tipo_objeto>::iterator var;
      // cria iterador 'var' para itens de 'tipo_objeto' em um vector

      // varre o contêiner e mostra cada item
for(var = conteiner.begin(); var != conteiner.end(); var++)
    cout<<"Item armazenado: "<< *var <<endl;
```

Vector

- tipo de contêiner sequencial, baseado em *arrays* (contêiner sequencial de tamanho fixo).
- suporta iteradores de acesso aleatório, que são normalmente implementados como ponteiros para os elementos de um vetor.
- vetores desta classe podem ser de tipos primitivos (inteiros, strings, pontos flutuante), bem como de tipos definidos pelo usuário (classes).
- esta estrutura de dados trabalha com posições de memória contíguas, logo o acesso direto a seus elementos também pode ser feito através do operador subscrito [ ].

Para usar os recursos desta classe, basta inserir o cabeçalho <**vector**> no código.
Para criar um objeto **vector**, usa-se: **vector<tipo_do_objeto> nome_do_objeto;**

As operações frequentemente utilizadas são:

- push_back(elemento) – inclui no final
- pop_back() – retira último elemento
- insert(posição, elemento) – insere o elemento na posição (deve ser um iterador)
- erase(posição) – exclui a posição
- clear() – zera o conteiner
- empty() – testa se vazio
- size() – retorna tamanho
- begin() – iterador de inicio
- end() – iterador de final

O exemplo abaixo apresenta um simples programa que usa esta classe com acesso aos elementos do vetor através do operador [ ].

```
#include <iostream>
#include <vector>
using namespace std;
int main(){
  vector<int> meuVetor;  // cria um vetor de inteiros vazio

  if (meuVetor.empty())  // testa se o vetor está vazio
    cout << "Vetor vazio!" << endl;
  else
```

```
    cout << "Vetor com elementos!" << endl;

  meuVetor.push_back(7); // inclue no fim do vetor um elemento
  meuVetor.push_back(11);
  meuVetor.push_back(2006);
  // vai imprimir três elementos {7, 11, 2006}
  for (int i = 0; i < meuVetor.size(); i++)
    cout << "Imprimindo o vetor...: " << meuVetor[i] << endl;

  cout << endl;
  meuVetor.pop_back();  // retira o último elemento

  // agora, só vai imprimir dois {7, 11}
  for (int i = 0; i < meuVetor.size(); i++)
    cout << "Meu vetor, de novo...: " << meuVetor[i] << endl;

  system("PAUSE");
  return 0;
}
```

Este exemplo percorre os elementos de um vector um iterador. Nota: quando usar iteradores, utilize o operador **!=** e a função **end()** para testar o fim do contêiner.

```
#include <iostream>
#include <vector>
using namespace std;
int main(){
  vector<int> meuVetor;  // cria um vetor de inteiros vazio
  vector<int>::iterator j;  // cria um iterador de inteiros

  meuVetor.push_back(7); // inclue no fim do vetor um elemento
  meuVetor.push_back(11);
  meuVetor.push_back(2006);

  // vai imprimir 3 elementos {7, 11, 2006}
  for(j = meuVetor.begin(); j != meuVetor.end(); j++ )
    cout << "Imprimindo o vetor...: " << *j << endl;
  cout << endl;

  // insere 55 como 2o elemento, deslocando os demais p/próx posição
  meuVetor.insert( meuVetor.begin() + 1, 55);

  // agora, imprimir 4 elementos {7, 55, 11, 2006}
  for(j = meuVetor.begin(); j != meuVetor.end(); j++ )
    cout << "Inseri no meio do vetor..: " << *j << endl;
  cout << endl;

  // retira 11 da lista (3a posição)
  meuVetor.erase( meuVetor.begin() + 2);

  // agora, tem que imprimir 3 de novo {7, 55, 2006}
  for(j = meuVetor.begin(); j != meuVetor.end(); j++ )
    cout << "Retirei no meio do vetor..: " << *j << endl;

  meuVetor.clear();  // limpa todo o vetor

  return 0;
}
```

Ordenação de Vector - para ordenar um vector com tipos primitivos (int, float, char e double), usa-se a função **`sort`**, conforme o trecho de código abaixo:

```
#include <iostream>
#include <algorithm> // para usar o sort
#include <vector>
using namespace std;
int main(){
    vector <float> V;

    V.push_back(-4);
    V.push_back(4);
    V.push_back(-9);
    V.push_back(-12);
    V.push_back(40);

    cout << "IMPRIMINDO..." << endl;
    for(int i=0; i<V.size(); i++)
        cout << V[i] << endl;

    sort(V.begin(), V.end());
    cout << "IMPRIMINDO EM ORDEM..." << endl;
    for(int i=0; i<V.size(); i++)
        cout << V[i] << endl;
    cout << "Fim..." << endl;
    return 0;
}
```

List

- tipo de contêiner sequencial que trabalha com operações de inserção e exclusão de elementos em qualquer posição do contêiner
- é implementada como uma lista duplamente encadeada.
- suporta iteradores de acesso bidirecional, o que permite percorrer uma lista para frente ou para trás => logo são necessários iteradores (o operador [ ] não é suportado por **`list`**).

Assim como vetores, listas desta classe podem ser de tipos de dados primitivos (inteiros, strings, pontos flutuante), bem como de tipos definidos pelo usuário (classes)
Para usar os recursos desta classe, basta inserir o cabeçalho **`<list>`** no código.
Para criar um objeto **`list`**, usa-se **`list<tipo_objeto> nome_do_objeto`**.

As operações frequentemente utilizadas são:

- push_back(elemento) – inclui no final
- push_front(elemento) – inclui no início
- pop_back() – retira último elemento
- pop_front() – retira 1º elemento
- insert(posição, elemento)
- erase(posição)
- remove(elemento)
- unique() – remove elementos duplicados
- find(início, fim, elemento) – procura elemento em um intervalo da lista, usando iteradores, retorno é um iterador
- sort() – ordena em ordem ascendente
- clear()
- empty()
- size()
- begin()
- end()

O exemplo a seguir mostra um simples programa usando listas e as operações apresentadas.

```
#include <iostream>
#include <list>
#include <algorithm>  // para usar o find
```

```
using namespace std;

int main(){
  list<double> minhaLista;   // cria uma lista de floats vazia
  list<double>::iterator k;  // cria um iterador de float

  minhaLista.push_back(7.5);
  minhaLista.push_back(27.26);
  minhaLista.push_front(-44);    // inserindo no início da lista
  minhaLista.push_front(7.5);    // inserindo no início da lista
  minhaLista.push_back(69.09);

  // vai imprimir seis elementos {7.5, -44, 7.5, 27.26, 69.09}
  for(k = minhaLista.begin(); k != minhaLista.end(); k++)
    cout << "Imprimindo a lista...: " << *k << endl;
  cout << endl;

  // insere -2.888 como último elemento
  minhaLista.insert( minhaLista.end(), -2.888);
  // retira o elemento -44 da lista
  minhaLista.remove(-44);
  // remove elementos duplicados da lista (no caso, 7.5 aparece 2x)
  minhaLista.unique();
  // ordena a lista, em ordem ascendente
  minhaLista.sort();

  // agora, mostra 4 elementos {-2.888, 7.5, 27.26, 69.09}
  for(k = minhaLista.begin(); k != minhaLista.end(); k++)
    cout << "Lista final ordenada...: " << *k << endl;
  cout << endl;

  // para usar find, informe ponto inicial e final de procura,
  // e o elemento, o método devolve um iterador para o objeto
  k = find(minhaLista.begin(), minhaLista.end(), 27.26);
  if( *k == 27.26 ) cout << "Elemento 27.26 encontrado!!!" << endl;
  else cout << "Nao existe o elemento procurado!!!" << endl;

  if (minhaLista.empty())
    cout<<"Lista vazia!"<<endl;
  else
    cout<<"Lista com "<< minhaLista.size() <<" elementos!"<<endl;

  minhaLista.clear();  // limpa toda a lista
  system("PAUSE");
  return 0;
}
```