ESCOLA POLITÉCNICA

Página diagramada pelo Estagiário de Docência Rafael Rieder

STL - Standard Template Library

Prof. Márcio Sarroglia Pinho

Definições

A STL C++ é uma biblioteca padronizada de funções, que oferece ao desenvolver um conjunto de classes de uso genérico, descrevendo contêineres (estruturas de dados, como pilhas, listas e filas), iteradores (objetos que percorrem elementos de um conjunto) e algoritmos básicos (principalmente os destinados a busca e classificação).

A STL C++ é baseada na STL desenvolvida pela SGI (http://www.sgi.com/tech/stl/, e seus componentes baseiam-se fortemente no uso de templates.

Uma das principais vantagens do uso desta biblioteca está na simplificação do trabalho com estruturas de dados, uma vez que o código baseado em ponteiros é complexo e exige atenção do desenvolvedor em testes e depuração. Vetores e listas, por exemplo, podem mudar de tamanho dinamicamente, e serem atribuídos um ao outro.

Iteradores

Os iteradores, similares a ponteiros, são usados para apontar para os elementos dos contêineres. Eles armazenam a informação aos tipos específicos de contêineres que eles operam. Isso significa dizer que eles devem ser implementados com o mesmo tipo do contêiner a percorrer.

Contêineres oferecem os métodos **`begin()`** e **`end()`** para o trabalho com iteradores. O operador `*` é usado para acessar o elemento apontado.

Assim, para criar e usar um iterador:

```
std::vector<tipo_do_objeto>::iterator var;  // cria um iterador 'var' para objetos 'tipo_do_objeto'

for ( var = conteiner.begin(); var != conteiner.end(); var++ ) // percorre o conteiner
  cout << "Imprime objeto armazenado no contêiner...: " << *var << endl;
```

Os próximos exemplos, com contêineres **`vector`**, **`list`** e **`deque`** apresentarão com detalhes o uso de iteradores.

Vectors

`Vector` é um tipo de contêiner seqüêncial, baseado em *arrays*. Ele suporta iteradores de acesso aleatório, que são normalmente implementados como ponteiros para os elementos de um vetor.

Vetores desta classe podem ser de tipos de dados primitivos (inteiros, strings, pontos flutuante), bem como de tipos definidos pelo usuário (classes).

Como esta estrutura de dados trabalha com posições de memória contíguas, 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: **`std::vector<tipo_do_objeto> nome_do_objeto`**.

As operações freqüentemente utilizadas são: **`push_back(elemento)`**, **`pop_back()`**, **`insert(posição, elemento)`**, **`erase(posição)`**, **`clear()`**, **`empty()`**, **`size()`**, **`begin()`** e **`end()`**.

O exemplo abaixo apresenta um exemplo simples de programa, usando esta classe com acesso aos elementos do vetor através do operador [ ]. O código-fonte pode ser obtido aqui.

```
#include <iostream>
#include <vector>
using std::cout;
using std::cin;
using std::endl;

int main()
{
  std::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. O código-fonte pode ser obtido aqui.

```
#include <iostream>
#include <vector>
using std::cout;
using std::cin;
using std::endl;

int main()
{
  std::vector<int> meuVetor;  // cria um vetor de inteiros vazio
  std::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 três elementos {7, 11, 2006}
  for ( j = meuVetor.begin(); j != meuVetor.end(); j++ )
    cout << "Imprimindo o vetor...: " << *j << endl;

  cout << endl;
  // insere 55 como segundo elemento, deslocando os demais para a próxima posição
  meuVetor.insert( meuVetor.begin() + 1, 55);

  // agora, imprimir quatro 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 (terceira posição)
  meuVetor.erase( meuVetor.begin() + 2);

  // agora, tem que imprimir três 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 como int, float, char e double, chame a função **sort** da STL, conforme o trecho de código abaixo:

```
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;
}
```

Vector com Classes

Além de armazenar tipos primitivos como int, float, char e double, as classes da STL permitem que se criem estruturas de dados a partir de classes ou structs definidadas pelo programador.

Para armazenar objetos de uma classe Pessoa em um vector, veja o exemplo abaixo.

```
#include <iostream>
using namespace std;
#include <algorithm>
#include <vector>

class Pessoa {
    string nome;
    int idade;
    public:
        Pessoa(string no, int id)
        {
            idade = id;
            nome = no;
        }
        string getNome()
        {
            return nome;
        }
        int getIdade()
        {
            return idade;
        }
};

int main(){

      vector <Pessoa> VP;
      vector <Pessoa>::iterator ptr;

      VP.push_back(Pessoa("Joao", 25));
      VP.push_back(Pessoa("Maria", 32));
      VP.push_back(Pessoa("Carla", 4));
      VP.push_back(Pessoa("Abel", 30));

      // percorrendo a lista com indices
      for(int i = 0; i < VP.size(); i++)
      {
            cout << "Nome: " << VP[i].getNome();
            cout << " - Idade: " << VP[i].getIdade() << endl;
      }
}
```

Ordenação de Vector com Classes

Para ordenar um vector criado a partir de uma classe, deve-se utilizar a função sort da STL. Neste caso, como há várias possibilidades de ordenação (pelos vários atributos da classe), é necessário criar uma função de comparação entre dois objetos da classe que forma o vector.

Esta função deve receber, por parâmetro, dois objetos (A e B) da classe que dá origem ao vector e deve retornar

true caso o objeto A venha ANTES do objeto B segundo o critério de oirdenação. Caso contrário, deve retirnar **false**.

No exemplo a seguir, a função de comparação compara dois objetos da classe **Pessoa**, com base no atributo nome. No caso a função define a ordem decrescente de nome.

```
#include <iostream>
using namespace std;
#include <vector>

.......
.......
........

bool ordena_por_nome(Pessoa A, Pessoa B)
{
    if (A.getNome() > B.getNome())
       return true;
    return false;
}

int main(){

      vector <Pessoa> VP;
      vector <Pessoa>::iterator ptr;

      VP.push_back(Pessoa("Joao", 25));
      VP.push_back(Pessoa("Maria", 32));
      VP.push_back(Pessoa("Carla", 4));
      VP.push_back(Pessoa("Abel", 30));


      sort ( VP.begin(), VP.end(), ordena_por_nome);

      // percorrendo a lista com um ITERATOR
      for(ptr = VP.begin(); ptr != VP.end(); ptr++)
      {
            cout << "Nome: " << ptr->getNome();
            cout << " - Idade: " << ptr->getIdade() << endl;
      }
    system("pause");
}
```

EXCEÇÕES

Para tratar exceções com VECTOR, consulte http://anaturb.net/C/out_of_range.htm

Lists

`List` é um tipo de contêiner seqüêncial que trabalha com operações de inserção e exclusão de elementos em qualquer posição do contêiner, e é implementada como uma lista duplamente encadeada. Ele suporta iteradores de acesso bidirecional, o que permite percorrer uma lista para frente ou para trás. Para tanto, 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 vector, usa-se: **`std::list<tipo_do_objeto> nome_do_objeto`**.

As operações freqüentemente utilizadas são: **`push_back(elemento)`**, **`push_front(elemento)`**, **`pop_back()`**, **`pop_front()`**, **`insert(posição, elemento)`**, **`erase(posição)`**, **`remove(elemento)`**, **`find(inicio, fim, elemento)`**, **`unique()`**, **`sort()`**, **`clear()`**, **`empty()`**, **`size()`**, **`begin()`** e **`end()`**.

O exemplo abaixo mostra um exemplo simples de programa usando listas e as operações apresentadas.
O código-fonte pode ser obtido aqui.

```
#include <iostream>
#include <list>
#include <algorithm>  // necessário para o método find
using std::cout;
using std::cin;
using std::endl;

int main()
{
  std::list<double> minhaLista;   // cria uma lista de floats vazia
  std::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 121.38 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, tem que imprimir quatro 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 o ponto inicial e final de procura, mais o elemento
  // este método STL devolve um iterador (ponteiro) para o objeto.
  k = find(minhaLista.begin(), minhaLista.end(), 27.26);
  if( *k == 27.26 ) cout << "Elemento 27.26 encontrado!!!" << endl;
  else cout << "Não 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;
}
```

Deques

`Deque` (pronuncia-se "*deek*") é um tipo de contêiner seqüêncial que fornece recursos de um **`vector`** e de uma **`list`** em um mesmo objeto. Ele fornece acesso a seus elementos através de iteradores e do operador [] (como um **`vector`**), e permite ler e modificar os elementos da mesma forma que uma **`list`** (no entanto, alguns métodos não estão disponíveis).

Deques mantém uma estrutura do tipo FIFO, e são comumente usadas para implementar filas circulares.
Um exemplo simples de **`deque`** pode ser obtido aqui.

Outros Exemplos

Lista de strings, com ordenação (usando objetos do tipo CPessoa)
Lista sobrecarregando operadores (usando objetos do tipo CPessoa)
Lista e funções polimórficas (usando objetos do tipo Ponto e Círculo)