Guia de Consulta Rápida

C++ STL

Joel Saade

Novatec Editora

ISBN: 85-7522-082-9

Primeira impressão: Fevereiro/2006

Novatec Editora Ltda.
Rua Luís Antônio dos Santos 110
02460-000 São Paulo Brasil
Tel.: +55 11 6959-6529
Fax: +55 11 6950-8869
E-mail: novatec@**novatec**.com.br
Site: www.**novatec**.com.br

Convenções

Convenção	Significado
ItInp	Iterador input.
ItOut	Iterador output.
ItFwd	Iterador forward.
ItBid	Iterador bidirectional.
ItRac	Iterador random access.
método	O mesmo que função-membro.
T	Tipo dos valores armazenados em um container (por exemplo, **int**, **float**, **char**).
X	Classe-container que contém objetos do tipo **T** (por exemplo, **vector<int>**).
a	Um objeto da classe **X** (por exemplo, **vector<int> a**).

Definições

Termo	Significado
[first, last)	Par de iteradores que define um intervalo iniciado em **first** e terminado, mas sem incluir **last**: intervalo fechado em **first** e aberto em **last**.
função unária	Função que recebe um argumento.
função binária	Função que recebe dois argumentos.
iteração	O ato de um iterador percorrer um container seqüencialmente, elemento a elemento.
método binário	Método que recebe dois argumentos.
método unário	Método que recebe um argumento.
past-the-end	Posição após a última em um container, assumida por um iterador.
seqüência	Um par de iteradores passado como argumento a um algoritmo que define um intervalo de elementos sobre o qual o algoritmo atuará.

Observações

- Para os programas-exemplo que utilizam as notações **it->first** e **it->second**, caso o compilador que você esteja usando não as aceite, utilize **(*it).first** e **(*it).second**.
- Nos programas-exemplo, as linhas de código **#include <iostream>** e **using namespace std;** foram suprimidas por serem comuns a todos os programas. No site da Novatec Editora, tais programas estão na sua forma integral, disponíveis para download (www.novatec.com.br/download/).

Introdução à STL

A Standard Template Library (STL) é uma biblioteca integrada à biblioteca padrão de C++, por meio do mecanismo de templates. Além de programas contém outros componentes, entre eles, destacam-se estruturas de dados e ponteiros "inteligentes". Na terminologia STL, as estruturas de dados são chamadas de containers, os programas são chamados de algoritmos e os ponteiros "inteligentes", de iteradores.

Os containers armazenam valores de um dado tipo (int, float etc). Os algoritmos correspondem às ações a serem executadas sobre os containers (ordenação, pesquisa e outras). Os iteradores percorrem os elementos dos containers da mesma forma que um índice percorre os elementos de um array. Esses três componentes são básicos e o seu conhecimento é o mínimo para se usar a STL.

A STL é uma caixa de ferramentas que auxilia, que traz soluções para muitos problemas de programação que envolvem estruturas de dados.

Bem, mas quais seriam essas estruturas de dados e quais seriam esses problemas?

As estruturas de dados básicas são: vetor, lista, deque, set, multiset, map e multimap. E mais as seguintes, criadas a partir das estruturas básicas: pilha, fila e fila com prioridade.

Entre os problemas encontrados no desenvolvimento de aplicações que usam estruturas de dados, destacam-se:

- ordenar os elementos de uma lista;
- efetuar uma pesquisa eficiente em um vetor;
- gerenciar a memória quando da inserção ou eliminação de elementos de uma deque.

A STL foi desenvolvida por Alexander Stepanov e Meng Lee, nos laboratórios da Hewlett Packard, com a valiosa colaboração de David R. Musser. Em 1994, o comitê ISO/ANSI C++ decidiu incorporar a STL à biblioteca padrão de C++ e neste mesmo ano, a HP disponibilizou a STL na Internet.

Para mostrar a poder da STL, veja o exemplo seguinte.

```
1   // Cria um vetor com nomes e o ordena ascendentemente
2 #include <algorithm>    // Para algoritmos
3 #include <vector>       // Para containers vector
4 #include <string>       // Para objetos string
5 int main()
6 {
7       vector<string> nomes; // Vetor de objetos string
8       // Iterador para vetores de objetos string
9       vector<string>::iterator it;
10      // Insere valores no vetor com o método push_back()
11      nomes.push_back("Ana");
12      nomes.push_back("Rose");
13      nomes.push_back("Jane");
14      nomes.push_back("Carla");
15      cout << "Vetor antes da ordenação:  ";
16      for (it = nomes.begin();it != nomes.end();++it)
17        cout << *it << " ";
```

```
18      cout << endl;
19      sort(nomes.begin(),nomes.end()); // Ordena o vetor
20      cout << "Vetor depois da ordenação: ";
21      for (it = nomes.begin();it != nomes.end();++it)
22        cout << *it << " ";
23      return 0;
24    }
```

Resultado da execução do programa

```
Vetor antes da ordenação:  Ana Rose Jane Carla
Vetor depois da ordenação: Ana Carla Jane Rose
```

As observações seguintes aplicam-se ao programa-exemplo recém-apresentado, sendo válidas também para a maioria dos programas-exemplo contidos neste guia, por apresentarem conceitos básicos importantes.

Linha	Descrição
2	É necessária para a maioria dos algoritmos.
3	É necessária para o uso do container **vetor**, havendo um arquivo-cabeçalho para cada container.
7	Declara um container do tipo **vetor** com elementos do tipo **string**, chamado **nomes**.
9	Declara um **iterador**, chamado **it**, para percorrer os elementos do vetor. Este iterador poderá ser utilizado com outros vetores desde que tenham elementos do tipo **string**.
11 a 14	Utilizam o método **push_back()** definido para vetores, cuja tarefa é inserir elementos no final de um vetor.
16	Acessa cada elemento do vetor, do primeiro até o último, por meio do iterador **it**.
17	Exibe o valor de cada elemento do vetor por meio do operador de de-referência *, que retorna o valor apontado pelo iterador **it**.
19	Utiliza o algoritmo **sort** para ordenar elementos de containers. Há de se notar que apenas uma linha de comando foi necessária para efetuar a ordenação do vetor.

Containers

São estruturas de dados que armazenam valores. Estão divididos em duas categorias: **seqüenciais** e **associativos classificados**, com as seguintes características:

- Alocação e gerenciamento da memória, por meio de um objeto alocador de memória próprio.
- Fornecimento de métodos para a sua manipulação.
- Reversibilidade: cada container pode ser acessado de forma reversa, isto é, do último para o primeiro elemento.

Tipos comuns a todos os containers

Tipo	Descrição
X::allocator_type	Representa o alocador de memória para **X**. Equivale ao argumento **Allocator**.
X::difference_type	Inteiro sinalizado que representa a diferença entre dois iteradores de **X**.
X::const_iterator	Iterador constante para acessar os elementos armazenados em um **X** constante.
X::iterator	Iterador para acessar os elementos armazenados em **X**.
X::const_reverse_iterator	Iterador constante para acessar, de forma reversa, os elementos armazenados em um **X** constante.
X::reverse_iterator	Iterador para acessar, de forma reversa, os elementos armazenados em **X**.
X::size_type	Inteiro sem sinal que armazena valores do tipo **difference_type** ou o tamanho de **X**.
X::value_type	Tipo dos elementos (**T**) armazenados em **X**. Para **map** e **multimap** é **pair<const Key,T>**, em que o primeiro membro do par é **const** para que a chave não seja alterada.

Containers seqüenciais

São containers cujos elementos estão em uma ordem linear. São homogêneos, pois armazenam valores de um mesmo tipo. Os tipos podem ser os básicos (**int**, **float** etc) ou criados pelo programador (estrutura ou classe). Um array de C++ é um exemplo de container seqüencial. Dispõem de gerenciamento automático de memória que permite ao tamanho do container variar dinamicamente, aumentando ou diminuindo à medida que elementos são inseridos ou eliminados. Os containers seqüenciais são: **vetor**, **deque** e **lista**.

Container	Arquivo-cabeçalho
vetor	vector
deque	deque
lista	list

Vetor

A classe-template **vector** representa uma estrutura de dados que pode ser comparada a um array dinâmico de C++, em que os elementos são armazenados de forma contígua e podem ser acessados aleatoriamente por meio do operador **[ ]**.

Iterador suportado: **random access**.

Declaração da classe

```
template <typename T,typename Allocator = allocator<T> > class vector
```

Definição de tipos

As classes-template **vector**, **deque** e **list** definem os tipos:

Tipo	Descrição
const_pointer	Ponteiro **const** para os elementos de vetores, deques e listas (**const T***).
pointer	Ponteiro para os elementos de vetores, deques e listas (**T***).
const_reference	Referência **const** para os elementos de vetores, deques e listas (**const T&**).
reference	Referência para os elementos de vetores, deques e listas (**T&**).

Construtores

vector()

```
vector(const Allocator& = Allocator());
```

Cria um vetor vazio.

Exemplo

```
#include <vector>
int main() {
  vector<int> v1;    // vetor de inteiros
  return 0; }
```

vector()

```
explicit vector(size_type n,const T& valor = T(),
    const Allocator& = Allocator());
```

Cria um vetor com **n** elementos e o inicializa com **valor**.

```
/* jvcria01.cpp - Cria um vetor inicializado com o valor 9.
  É exibido com a notação de array e com iteradores */
#include <vector>
int main() {
  vector<int> v1(3,9);
  vector<int>::iterator it;   // Iterador
  cout << "Vetor v1: ";
  for (int i = 0;i < 3;++i) cout << v1[i] << " ";
  cout << "  Vetor v1: ";
  for (it = v1.begin();it != v1.end();++it) cout << *it << " ";
  return 0; }
```

vector()

```
vector(const vector<T, Allocator>& x);
```

Cria um vetor e o inicializa com os elementos do vetor **x**.

```
/* jvconcop.cpp - Cria o vetor v2 a partir do vetor v1.
  Exibe os vetores com iteradores */
#include <vector>
int main() {
  vector<int> v1(3);
  vector<int>::iterator it; // Iterador
  // Atribui valores com a notação de array
  v1[0] = 1; v1[1] = 2; v1[2] = 3;
  cout << "Vetor v1: ";
  for (it = v1.begin();it != v1.end();++it) cout << *it << " ";
  vector<int> v2(v1);       // Cria o vetor v2
  cout << "  Vetor v2: ";
  for (it = v2.begin();it != v2.end();++it) cout << *it << " ";
  return 0; }
```

vector()

```
template <typename ItInp>
vector(ItInp first,ItInp last,const Allocator& = Allocator());
```

Cria um vetor, inicializado com os elementos do intervalo [**first, last**).

```
/*  jvint.cpp - Cria o vetor v2 com intervalo do vetor v1 */
#include <vector>
int main() {
  vector<int> v1(3);
  v1[0] = 1; v1[1] = 2; v1[2] = 3;
  cout << "Vetor v1: ";
  for (int i = 0;i < 3;++i) cout << v1[i] << " ";
  vector<int> v2(v1.begin(),v1.end());  // Cria v2
  cout << "  Vetor v2: ";
  for (int i = 0;i < 3;++i) cout << v2[i] << " ";
  return 0; }
```

Operações de atribuição e comparação

operator=()

```
vector<T,Allocator>& operator=(const vector<T,Allocator>& x);
```

Substitui o conteúdo do vetor corrente pelo conteúdo do vetor **x**.

```
// jvopeq.cpp
#include <vector>
int main() {
  vector<int> v1(3),v2(4);
  vector<int>::iterator it;    // Iterador
  v1[0] = 5; v1[1] = 6; v1[2] = 7;
  v2[0] = 1; v2[1] = 1; v2[2] = 1; v2[3] = 1;
  cout << "v1: ";
  for (it = v1.begin();it != v1.end();++it) cout << *it << " ";
  cout << "  v2 antes da substituição: ";
  for (it = v2.begin();it != v2.end();++it) cout << *it << " ";
  cout << endl;
  v2 = v1;
  cout << "v2 depois da substituição: ";
  for (it = v2.begin();it != v2.end();++it) cout << *it << " ";
  return 0; }
```

operator==()

```
template <typename T,typename Allocator>
bool operator==(const vector<T,Allocator>& x,
    const vector<T,Allocator>& y);
```

Retornará **true** se os vetores **x** e **y** tiverem o mesmo tamanho e os mesmos elementos na mesma ordem, caso contrário, **false**.

```
// jvopeqeq.cpp
#include <vector>
int main() {
  vector<int> v1(3),v2(3);
  v1[0] = 1; v1[1] = 2; v1[2] = 3;
  v2[0] = 1; v2[1] = 2; v2[2] = 3;
  bool res = (v1 == v2);
  if (res == true)
       cout << "Vetores iguais";
  else
       cout << "Vetores diferentes";
  return 0; }
```

operator<()

```
template <typename T,typename Allocator>
bool operator<(const vector<T,Allocator>& x,
    const vector<T,Allocator>& y);
```

Retornará **true** se o vetor **x** for lexicograficamente menor que o vetor **y**, caso contrário, **false**.

operator!=()

```
template <typename T,typename Allocator>
bool operator!=(const vector<T,Allocator>& x,
    const vector<T,Allocator>& y);
```

Retornará **true** se vetor **x** for diferente do vetor **y**, caso contrário, **false**.

operator>()

```
template <typename T,typename Allocator>
bool operator>(const vector<T,Allocator>& x,
    const vector<T,Allocator>& y);
```

Retornará **true** se o vetor **x** for maior que o vetor **y**, caso contrário, **false**.

operator<=()

```
template <typename T,typename Allocator>
bool operator<=(const vector<T,Allocator>& x,
    const vector<T,Allocator>& y);
```

Retornará **true** se o vetor **x** for menor que ou igual ao vetor **y**, caso contrário, **false**.

operator>=()

```
template <typename T,typename Allocator>
bool operator>=(const vector<T,Allocator>& x,
    const vector<T,Allocator>& y);
```

Retornará **true** se o vetor **x** for maior que ou igual ao vetor **y**, caso contrário, **false**.

Métodos

at()

```
reference at(size_type n);
const_reference at(size_type n) const;
```

Retorna uma referência (ou uma referência constante para um vetor constante) para o elemento da posição **n**.

```
/*  jvat.cpp - Retorna o valor de determinado elemento de
  um vetor */
#include <vector>
int main() {
  vector<int> v1(3);
  v1[0] = 1; v1[1] = 2; v1[2] = 3;
  vector<int>::reference val = v1.at(2);
  cout << "Valor da posição 2: " << val;
  return 0; }
```

capacity()

```
size_type capacity() const;
```

Retorna o número máximo de elementos que o vetor pode armazenar sem realocação de memória.

```
// jvcapac.cpp
#include <vector>
int main() {
  vector<int> v1(5);
  vector<int>::size_type qtd = v1.capacity();
  cout << "Máximo: " << qtd;
  return 0; }
```

operator[]()

```
reference operator[](size_type n);
const_reference operator[](size_type n) const;
```

Retorna uma referência (ou uma referência constante para um vetor constante) para o elemento da posição **n**.

```
// jvoparr.cpp
#include <vector>
int main() {
  vector<int> v1(3);
  v1[0] = 1; v1[1] = 2; v1[2] = 3;
  vector<int>::reference valor = v1[2];
  cout << "Valor da posição 2: " << valor;
  return 0; }
```

reserve()

```
void reserve(size_type n);
```

Altera o tamanho do vetor de modo que a memória seja gerenciada adequadamente. A realocação ocorrerá se a capacidade atual do vetor for menor que **n**. Após a chamada a **reserve()** a capacidade será maior que ou igual à **n** se a realocação ocorrer, caso contrário, será igual à capacidade anterior.

swap()

```
void swap(vector<T,Allocator>& x);
```

Troca o conteúdo do vetor chamador pelo conteúdo do vetor **x** e vice-versa.

```
// jvswap.cpp
#include <vector>
int main() {
  vector<int> v1(2),v2(2);
  v1[0] = 3; v1[1] = 3; v2[0] = 5; v2[1] = 5;
  v1.swap(v2);
  cout << "v1 depois de swap(): ";
  for (int i = 0;i < 2;++i) cout << v1[i] << " ";
  cout << "  v2 depois de swap(): ";
  for (int i = 0;i < 2;++i) cout << v2[i] << " ";
  return 0; }
```

Deque

A classe-template **deque** representa uma estrutura de dados do tipo **double-ended queue** (fila com duas extremidades). A exemplo do **vetor**, a **deque** também suporta o operador **[ ]**. As operações de inserção e eliminação podem ser feitas em ambos os extremos e os seus elementos não são armazenados em posições contíguas de memória.

Iterador suportado: **random access**.

Declaração da classe

```
template <typename T,typename Allocator = allocator<T> > class deque
```

Definição de tipos

A classe-template **deque** define tipos que são iguais aos definidos pela classe-template **vector**, pág. 8.

Construtores

deque()

```
deque(const Allocator& = Allocator());
```

Cria uma deque vazia.

Exemplo

```
#include <deque>
int main() {
  deque<int> d1;
  return 0; }
```

deque()

```
explicit deque(size_type n,const T& valor = T(),
    const Allocator& = Allocator());
```

Cria uma deque com **n** elementos e a inicializa com **valor**.

```
/* jdcria01.cpp -  Cria uma deque inicializada com o valor 9.
  É exibida com a notação de array e com iteradores */
#include <deque>
int main() {
  deque<int> d1(3,9);
  deque<int>::iterator it;    // Iterador
  cout << "Deque d1: ";
  for (int i = 0;i < 3;++i) cout << d1[i] << " ";
  cout << "  Deque d1: ";
  for (it = d1.begin();it != d1.end();++it) cout << *it << " ";
  return 0; }
```

deque()

```
deque(const deque<T,Allocator>& x);
```

Cria uma deque e a inicializa com os elementos da deque **x**.

```
// jdconcop.cpp - Cria uma deque (d2) a partir de outra (d1)
#include <deque>
int main() {
  deque<int> d1(3);
  deque<int>::iterator it;    // Iterador
  // Atribui valores com a notação de array
  d1[0] = 1; d1[1] = 2; d1[2] = 3;
  cout << "Deque d1: ";
  for (it = d1.begin();it != d1.end();++it) cout << *it << " ";
  deque<int> d2(d1);   // Cria a deque d2
  cout << "  Deque d2: ";
  for (it = d2.begin();it != d2.end();++it) cout << *it << " ";
  return 0; }
```

deque()

```
template <typename ItInp>
deque(ItInp first,ItInp last,const Allocator& = Allocator());
```

Cria uma deque, inicializada com os elementos do intervalo [**first, last**).

```
/*  jdint.cpp - Cria a deque d2 com intervalo da deque d1 */
#include <deque>
int main() {
  deque<int> d1(3);
  d1[0] = 1; d1[1] = 2; d1[2] = 3;
  cout << "Deque d1: ";
  for (int i = 0;i < 3;++i) cout << d1[i] << " ";
  deque<int> d2(d1.begin(),d1.end());   // Cria d2
  cout << "  Deque d2: ";
  for (int i = 0;i < 3;++i) cout << d2[i] << " ";
  return 0; }
```

Operações de atribuição e comparação

operator=()

```
deque<T,Allocator>& operator=(const deque<T,Allocator>& x);
```

Substitui o conteúdo da deque corrente pelo conteúdo da deque **x**.

```
// jdopeq.cpp
#include <deque>
int main() {
  deque<int>::iterator it;     // Iterador
  deque<int> d1(3),d2(4);
  d1[0] = 5; d1[1] = 6; d1[2] = 7;
  d2[0] = 1; d2[1] = 1; d2[2] = 1; d2[3] = 1;
  cout << "d1: ";
  for (it = d1.begin();it != d1.end();++it) cout << *it << " ";
  cout << "  d2 antes da substituição: ";
  for (it = d2.begin();it != d2.end();++it) cout << *it << " ";
  cout << endl;
  d2 = d1;
  cout << "d2 depois da substituição: ";
  for (it = d2.begin();it != d2.end();++it) cout << *it << " ";
  return 0; }
```

operator==()

```
template <typename T,typename Allocator>
bool operator==(const deque<T,Allocator>& x,
    const deque<T,Allocator>& y);
```

Retornará **true** se as deques **x** e **y** tiverem o mesmo tamanho e os mesmos elementos na mesma ordem, caso contrário, **false**.

```
// jdopeqeq.cpp
#include <deque>
int main() {
  deque<int> d1(3),d2(3);
  deque<int>::iterator it;     // Iterador
  d1[0] = 1; d1[1] = 2; d1[2] = 3;
  d2[0] = 1; d2[1] = 2; d2[2] = 3;
  bool res = (d1 == d2);
  if (res == true)
       cout << "Deques iguais";
  else
       cout << "Deques diferentes";
  return 0; }
```

operator<()

```
template <typename T,typename Allocator>
bool operator<(const deque<T,Allocator>& x,
    const deque<T,Allocator>& y);
```

Retornará **true** se a deque **x** for lexicograficamente menor que a deque **y**, caso contrário, **false**.

operator!=()

```
template <typename T,typename Allocator>
bool operator!=(const deque<T,Allocator>& x,
    const deque<T,Allocator>& y);
```

Retornará **true** se a deque **x** for diferente da deque **y**, caso contrário, **false**.

operator>()

```
template <typename T,typename Allocator>
bool operator>(const deque<T,Allocator>& x,
    const deque<T,Allocator>& y);
```

Retornará **true** se a deque **x** for maior que a deque **y**, caso contrário, **false**.

operator<=()

```
template <typename T,typename Allocator>
bool operator<=(const deque<T,Allocator>& x,
    const deque<T,Allocator>& y);
```

Retornará **true** se a deque **x** for menor que ou igual à deque **y**, caso contrário, **false**.

operator>=()

```
template <typename T,typename Allocator>
bool operator>=(const deque<T,Allocator>& x,
    const deque<T,Allocator>& y);
```

Retornará **true** se a deque **x** for maior que ou igual à deque **y**, caso contrário, **false**.

Métodos

at()

```
reference at(size_type n);
const_reference at(size_type n) const;
```

Retorna uma referência (ou uma referência constante para uma deque constante) para o elemento da posição **n**.

```
/*  jdat.cpp - Retorna o valor de determinado elemento de
   uma deque */
#include <deque>
int main() {
   deque<int> d1;
   // Insere valores com o método push_back()
   d1.push_back(1); d1.push_back(2); d1.push_back(3);
   deque<int>::reference valor = d1.at(2);
   cout << "Valor da posição 2: " << valor;
   return 0; }
```

operator[]()

```
reference operator[](size_type n) const;
const_reference operator[] (size_type n) const;
```

Retorna uma referência (ou uma referência constante para uma deque constante) para o elemento da posição **n**.

```
/*  jdoparr.cpp - Retorna o valor de determinado elemento de
   uma deque */
#include <deque>
int main() {
   deque<int> d1(3);
   d1[0] = 1; d1[1] = 2; d1[2] = 3;
```

```
  deque<int>::reference valor = d1[2];
  cout << "Valor da posição 2: " << valor;
  return 0; }
```

pop_front()

```
void pop_front();
```

Elimina o primeiro elemento da deque.

```
// jdpopfro.cpp
#include <deque>
#include <string>    // Para objetos string
int main() {
  deque<string> d1(2);
  deque<string>::iterator it;    // Iterador
  d1[0] = "alfa"; d1[1] = "beta";
  cout << "Antes de pop_front(): ";
  for (it = d1.begin();it != d1.end();++it) cout << *it << " ";
  d1.pop_front();
  cout << "  Depois de pop_front(): ";
  for (it = d1.begin();it != d1.end();++it) cout << *it << " ";
  return 0; }
```

push_front()

```
void push_front(const T& x);
```

Insere o elemento **x** no início da deque.

```
// jdpusfro.cpp
#include <deque>
#include <string>    // Para objetos string
int main() {
  deque<string> d1;
  deque<string>::iterator it;  // Iterador
  d1.push_back("alfa"); d1.push_back("beta");
  d1.push_back("gama");
  cout << "Antes de push_front():  ";
  for (it = d1.begin();it != d1.end();++it) cout << *it << " ";
  cout << endl;
  d1.push_front("delta");
  cout << "Depois de push_front(): ";
  for (it = d1.begin();it != d1.end();++it) cout << *it << " ";
  return 0; }
```

swap()

```
void swap(deque<T,Allocator>& x);
```

Troca o conteúdo da deque chamadora pelo conteúdo da deque **x** e vice-versa.

```
// jdswap.cpp - Troca o conteúdo de duas deques
#include <deque>
int main() {
  deque<int> d1,d2;
  deque<int>::iterator it;  // Iterador
  for (int i = 1;i < 5;i++) // Insere 1 2 3 4
       d1.push_back(i);
  for (int i = 5;i < 9;i++) // Insere 5 6 7 8
       d2.push_back(i);
  d1.swap(d2);
  cout << "d1 depois de swap(): ";
  for (it = d1.begin();it != d1.end();++it) cout << *it << " ";
  cout << "  d2 depois de swap(): ";
  for (it = d2.begin();it != d2.end();++it) cout << *it << " ";
  return 0; }
```