Estrutura de Dados

Aula

STL - Standard Template Library

Containers

C++ STL

A STL (Standard Template Library) é uma biblioteca de algoritmos e estruturas de dados genéricas, integrada à biblioteca padrão de C++ através de mecanismos de *templates*. Criada por Alexander Stepanov e Meng Lee (Hewlett-Packard), adicionada ao C++ em 1994.

Templates são implementações generalizadas para diferentes estruturas de dados. Sobre essa ferramenta, o STL nos fornece *containers* para armazenar e realizar operações numa coleção de objetos.

C++ STL - Containers

Os *containers* são indispensáveis na programação competitiva.
Exemplos de *containers*:

- Containers sequenciais: **vector**, **deque** e **list**
- Adaptadores de containers: **stack**, **queue** e **priority_queue**
- Containers associativos classificados: **set** e **map**

Container: deque

Deque (acrônimo de "double-ended queue") é, como o próprio nome nos indica, uma **fila com duas extremidades**. São um tipo de container de tamanho dinâmico que pode ser expandido ou diminuído nas duas extremidades (*front* e *back*).

Também permitem o **acesso direto** a todos os seus elementos (operador []).

Apresentam funcionalidade **similar ao *vector*, mas com inserção e remoção eficiente de elementos também no começo da sequência**, não somente no fim. Porém, os elementos não são necessariamente armazenados de forma contígua.

Referência completa:
http://www.cplusplus.com/reference/deque/deque/

Principais métodos:

Acesso a elemento:	operador **[]** **front()** **back()**
Iteradores:	**begin()** **end()**
Capacidade:	**size()** **empty()**
Modificadores:	**push_back** **push_front** **pop_back()** **pop_front()** **insert erase** **clear()**

Container: deque

```
#include <deque>
deque<int> dq1;                          // declara variável do tipo deque
dq1.push_front(1);                        // insere elemento no inicio do deque
dq1.push_front (2);
dq1.push_back (3);                       // insere elemento no final do deque
dq1.push_back(4);
dq1.begin();                             // acessa ao primeiro elemento do deque
dq1.end();                               // acessa ao último elemento do deque
deque<int> dq2 (dq1.begin() +1, dq1.end()-1);
                                         // cria novo deque copiando elementos de outro deque
dq1 [1] = 5;                             // atribui valor a posição 1 do deque
```

RESULTADO:

dq1 { }

dq1 {1}

dq1 {2;1}

dq1 {2;1;3}

dq1 {2;1;3;4}

{2}

{4}

dq2 {2;1;3;4}

dq1 {5;1;3;4}

Container: deque

```
dq1.erase(dq1.begin());                  // exclui um elemento do deque
dq1.insert (dq1.end()-1, 2, 6);          // insere elemento 6 duas vezes a partir do
                                            penúltimo índice do deque
sort (dq1.begin(), dq1.end());           // ordena o deque do primeiro ao último elemento
deque<int> dq3;
dq1.size();                              // retorna o tamanho do deque
dq3.resize (dq1.size() +dq2.size());     // redefine o tamanho do deque
merge (dq1.begin(), dq1.end(), dq2.begin(), dq2.end(), dq3.begin());
                                         // reúne os conteúdos de dois deques em outro
```

RESULTADO:

dq2 {2;1;3;4}

dq1 {5;1;3;4}

dq1 {1;3;4}

dq1 {1;3;6;6;4}

dq1 {1;3;4;6;6}

dq3 { }

dq1 5

dq3 8

dq3 {1;3;4;6;6; 2;1;3;4}

Container: vector

Representam o **mesmo tipo de estrutura de um *array* em C**, e podem ser manipulados com a **mesma eficiência**, mas mudam seu tamanho dinamicamente e automaticamente.

Neles, os elementos são armazenados de forma contígua e podem ser acessados aleatoriamente através do operador [].

Sua utilização é importante especialmente quando não conhecemos o tamanho do vetor com antecedência.

São bastante **eficientes no acesso aos elementos e na inserção e remoção de elementos no fim** (*push_back* e *pop_back*). Para operações de inserção e remoção de elementos em outros lugares, são piores que outras estruturas como *list* e *deque*.

Container: vector

Principais métodos (operações):

Acesso a elemento:	**operador []** **front() back()**
Iteradores:	**begin()** **end()**
Capacidade:	**size()** **empty()**
Modificadores:	**push_back** **pop_back()** **insert erase** **clear() resize**

```
using namespace std;

int main() {

    vector<int> A;       //vetor vazio de inteiros
    vector<int> B(4);    //vetor com 4 elementos
    vector<int> C(4,1); //4 elementos de valor 1

    A.push_back(16);           //insere 16 no fim do vetor
    A.insert(A.begin(),20);    //insere 20 no inicio do vetor
    printf("%d\n", A.size()); //imprime tamanho do vetor

    B.resize(3,0);   //altera B para conter 3 zeros
    B[0] = A.back(); //atribui a B[0] o último elemento de A

    while(!C.empty()) {      //enquanto C for não-vazio:
        C.erase(C.end()-1); //remove o último elemento
    }

    A.clear(); B.clear(); C.clear(); //remove todos os elementos
```

Referência:
http://www.cplusplus.com/reference/vector/vector/

Container: vector

Iteradores

Iteradores são ponteiros que contém o endereço de memória de um elemento do container.
Nos containers sequenciais, os elementos são armazenados contiguamente.

Para os principais iteradores de *vector*:

```
A.begin()       - Ponteiro para o primeiro elemento de A
A.end()         - Ponteiro para um elemento nulo após o último elemento de A
A.begin()+i     - Ponteiro para o i-ésimo elemento de A
*(A.begin()+i)  - Valor do i-ésimo elemento de A
```

Ponteiros: São variáveis que armazenam o endereço de memória de uma variável. Para acessar o valor da variável deve-se utilizar um `*` antes da variável.

Container: vector

Exercício:

1) Crie um vetor vazio e insira valores aleatórios de 0 a 9.

   Para gerar um número aleatório de 0 a 9:

```
int x = rand() % 10;
```

2) Para cada elemento do vetor, começando do primeiro elemento:
   - se for par, remova esse elemento;
   - se for ímpar, multiplique pelo tamanho do vetor.

1) Encontre a posição dos valores máximo e mínimo do vetor resultante.
2) Crie um novo vetor a partir do anterior com todos os valores entre o máximo e o mínimo, inclusive.
3) Imprima o vetor resultante em ordem crescente.

```
//Ordenar um vetor:
sort(A.begin(), A.end()); //em ordem crescente
sort(A.begin(), A.end(), less<int>());
sort(A.begin(), A.end(), greater<int>()); //decrescente
sort(A.begin(), A.begin()+3); //sub-vetor

//Vetor a partir de um sub-vetor:
vector<int> B(A.begin(), A.begin()+4);
vector<int> C(A.end()-4, A.end());
vector<int> D(A.begin()+2, A.end()-1);

//Maximo e minimo de um vetor
int maximo = *max_element(A.begin(), A.end());
int minimo = *min_element(A.begin(), A.end());
```

Container: list

São listas duplamente encadeadas, que permitem **operações de inserção e remoção de itens em tempo constante** em qualquer posição da sequência e iteração nos dois sentidos.

Armazenam elementos de maneira não contígua, mantendo a ordem internamente através de uma associação: cada elemento possui uma ligação com seu antecessor e sucessor na lista. **Não é possível utilizar o operador []**.

Em comparação com *deque* e *vector*, a *list* se sai **melhor nas operações de inserção, remoção** e movimentação de elementos em qualquer posição do container para a qual um iterador já tenha sido obtido, e, portanto, se destaca também em algoritmos que fazem uso intensivo destas operações, como algoritmos de ordenação.

Container: list

Principais métodos (funções):

Acesso a elemento:	**front()** **back()**
Iteradores:	**begin()** **end()**
Capacidade:	**size()** **empty()**
Modificadores:	**push_back** **push_front** **pop_back()** **pop_front()** **insert erase** **clear()** **resize**

```cpp
#include<bits/stdc++.h>

using namespace std;

int main() {

    list<string> A;     //lista vazia de string
    list<string>::iterator it; //iterador tipo lista de string

    A.push_back("Costa");          //insere "Roberto" no fim do lista
    A.insert(A.begin(),"Roberto"); //insere "Affonso" no inicio do lista

    it = A.begin(); //it recebe a primeira posicao de A
    it++;           //it recebe a segunda posicao de A
    A.insert(it, "Affonso"); //insere "Affonso" na posicao apontada por it

    for(it = A.begin(); it!=A.end(); it++) {
        cout<<*it<<endl;    //imprime o valor apontado por it
    }

    A.clear();  //remove todos os elementos

    return 0;
}
```

Referência:
http://www.cplusplus.com/reference/list/list/

Container: list

Iteradores

Na *list*, os elementos não são armazenados contiguamente. Em vez disso, os iteradores são **bidirecionais**, onde um iterador possui uma ligação com o próximo elemento e o elemento anterior.

Iteradores bidirecionais **só podem ser incrementados (++) ou decrementados (--)**. Para se percorrer uma lista, é necessário começar do primeiro (último) elemento e incrementar (decrementar) o ponteiro até a posição desejada.

```
list<int>::iterator it = A.begin(); - Iterador aponta para o primeiro elemento de A
it++; it++; it--;                   - it aponta para o 2° elemento de A
*it                          - Valor do 2° elemento de A
it += 2; it = it - 1;        - Não permitido. it não suporta essas operações.
```

Adaptador de Container: queue

queue é um adaptador de container projetado especificamente para operar em um contexto FIFO (*first in, first out*), onde **elementos são sempre inseridos no fim e removidos apenas do início** do container.

Exatamente como numa fila da vida real, **o primeiro a entrar é o primeiro a sair**.

Principais métodos:

empty()
size() front()
back()
push
pop()

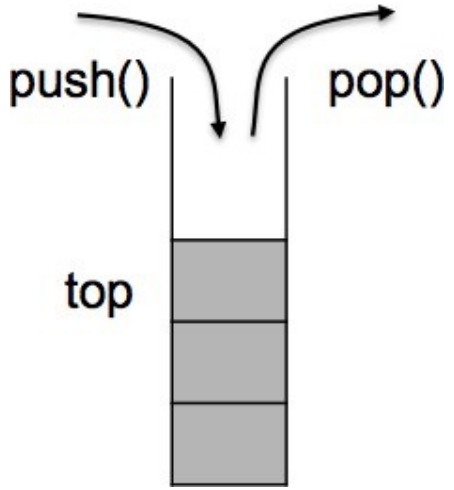
Referência completa:
http://www.cplusplus.com/reference/queue/queue/

Adaptador de Container: stack

stack (pilha) é um adaptador de container projetado especificamente para operar em um contexto LIFO (*last in, first out*), onde elementos são SEMPRE inseridos e removidos apenas do final (topo) do container.
Exatamente como uma pilha da vida real: o último a entrar é o primeiro a sair.

Principais métodos:

empty()
size()
top()
push
pop(
)



Referência completa:
http://www.cplusplus.com/reference/stack/stack/

Estrutura de Dados

Aula

STL - Standard Template Library

Containers