Curso de C++ para programação de Jogos

Biblioteca STL

Por: Vitor Maia
www.dcc.ufrj.br/~vitormaia

STL significa Standard Template Library. É uma biblioteca que contém algumas estruturas de dados, como árvore binária e lista encadeada.

Estas estruturas são chamadas de Containers. Elas armazenam coleções de elementos, assim como faz um vetor, mas de modo mais otimizado e especializado.

Os Containers da STL são:
vector, deque, list, set, multiset, map,
multimap, bitset, stack, queue e priority_queue.

Serão apresentados apenas:
vector, deque, stack e queue.
As outras estruturas possuem detalhes mais avançados de C++, e/ou dificilmente são usadas em programação de jogos. Para conhecê-las:
http://www.cplusplus.com/reference/stl/

Stack - Pilha

Imagine o seguinte caso: Você está lavando pratos na cozinha da sua casa. Cada prato, após lavado, é momentâneamente deixado um sobre o outro, em uma pilha. Após ter sido lavada uma certa quantidade, você nota que a pilha já está muito grande e resolve guardar os pratos já lavados, para então lavar os que ainda ficaram sujos. Qual prato você tira da pilha primeiro? O que colocou por último, correto?

Stacks funcionam como pilhas de pratos. Elementos são apenas inseridos ou tirados, e o elemento tirado é sempre o que foi inserido por último.

Se numa stack de números inteiros forem adicionados os números: 6, 4, 8, 1, 9,
os elementos são retirados obrigatoriamente na ordem: 9, 1, 8, 4, 6.

O tipo stack<T> possui os seguintes métodos:

empty() : retorna true caso a pilha esteja vazia;
size() : retorna o número de elementos da pilha;
top() : acessa o elemento no topo da pilha;
push(T) : adiciona elemento no topo da pilha;
pop() : retira o elemento do topo.

>> Os Ts acima significam "qualquer tipo".
Uma pilha pode ser de ints, floats, etc,
ou de qualquer tipo de objeto.

Código Exemplo

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

int main (int argc, char **argv) {

    // Declaração da pilha de inteiros.
    stack<int> s;

    // Elementos adicionados na pilha.
    s.push(6);
    s.push(4);
    s.push(8);
    s.push(1);
    s.push(9);

    cout << "size(): " << s.size() << endl;
    cout << "Imprimindo pilha: ";
    while(!s.empty()) {
        cout << s.top() << " ";
        s.pop();
    }
    cout << endl << "size(): " << s.size() << endl;
    return 0;
}
```

Queue - Fila

Assim como o tipo stack, o tipo queue pode ser também comparado a entidades cotidianas. Pense no funcionamento de uma fila qualquer, como a de um banco. Quem chega por último entra no fim da fila. É atendido primeiro quem chegou antes.

O tipo queue funciona como uma fila comum.
São inseridos elementos, como numa pilha, mas os primeiros a sair são os que estão no início.

Se numa queue de números inteiros forem adicionados os números: 6, 4, 8, 1, 9,
os elementos são retirados obrigatoriamente na ordem: 6, 4, 8, 1, 9.
Ou seja, eles saem na mesma ordem que entraram.

O tipo queue<T> possui os seguintes métodos:

empty() : retorna true caso a fila esteja vazia;
size() : retorna o número de elementos da fila;
front() : acessa o primeiro elemento da fila;
back() : acessa o último elemento da fila;
push(T) : adiciona elemento no fim da fila;
pop() : retira o elemento do início da fila.

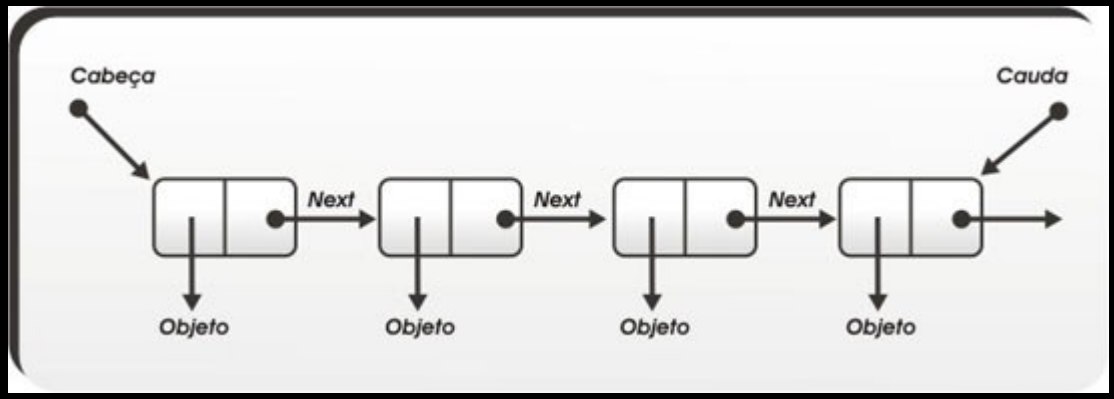
>> Os Ts acima significam o mesmo que na stack: "qualquer tipo".

Código Exemplo

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

int main (int argc, char **argv) {

    queue<int> q;

    q.push(6);
    q.push(4);
    q.push(8);
    q.push(1);
    q.push(9);

    cout << "size(): "  << q.size()  << endl;
    cout << "front(): " << q.front() << endl;
    cout << "back(): "  << q.back()  << endl;
    cout << "Imprimindo fila: ";
    while(!q.empty()) {
        cout << q.front() << " ";
        q.pop();
    }
    cout << endl << "size(): " << q.size() << endl;
    return 0;
}
```

Listas encadeadas

Os containers deque e vector são *similares* a uma estrutura de dados chamada lista encadeada. Antes de mostrá-los, descreverei as características desta estrutura.



Listas encadeadas são estruturas de dados similares a vetores, mas que possuem algumas facilidades para a manipulação dos elementos.

Características:

- Listas não possuem um tamanho limite, nem índices. Podem ir crescendo durante a inserção de elementos.

- Um elemento pode ser inserido no meio da lista. Este "encaixe" no meio da lista empurra os elementos seguintes para a frente.

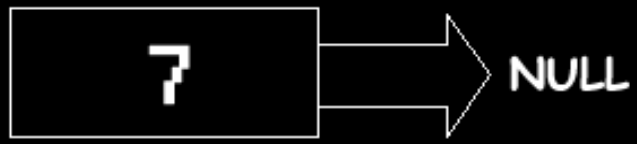
- A inserção empurra os elementos, e analogamente, a remoção de um elemento no meio da lista traz os elementos de volta para a frente. Esta liberdade não existe com vetores.

- Um elemento inserido fica armazenado em um objeto chamado de "Nó". Um nó armazena um dado e possui um ponteiro para um próximo nó.

Estas são características de uma lista encadeada simples. Existem outros tipos: lista duplamente encadeada, lista circular, lista ordenada... Para aprender os detalhes, há um tópico excelente na Wikipedia:

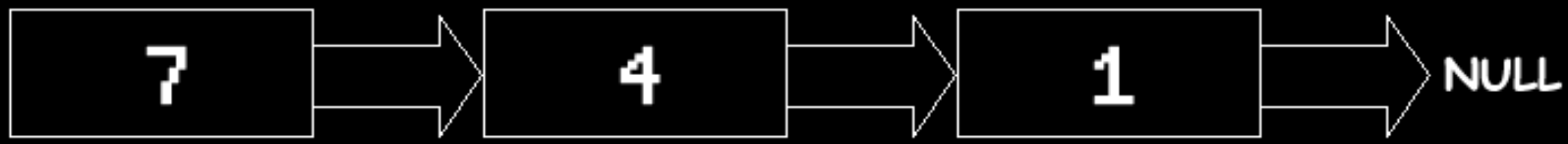
http://en.wikipedia.org/wiki/Linked_list

Lista recém criada, com um elemento apenas.
O ponteiro para o próximo aponta para NULL.



Elemento inserido no fim.

O elemento 4 foi inserido na segunda posição.
Veja que o nó contendo o 1 foi
simplesmente empurrado, e caso existissem nós seguintes, andariam também.



O nó com o elemento 4 foi removido.
O nó com 1 e seus seguintes, caso houvesse,
dariam um passo para o inicio também.

vector<T>

Vector é um container muito parecido com uma lista encadeada, mas que possui uma diferença: possui índices. É possível acessar elementos usando o operador [], além de ter todas as facilidades de adição/remoção de uma lista encadeada. Estão listados abaixo alguns métodos:

size() -> retorna o número de elementos;
resize(x) -> modifica o tamanho alocado para x unidades;
empty() -> confere se o vector está vazio;
begin() e end() -> iteram o vector;
at(i) -> acessa a posição i;
front e back() -> acessam o primeiro e o último elementos;
erase(i) -> remove um elemento específico, ou uma sequencia deles;
push_back(T) -> adiciona o elemento T no fim;
pop_back() -> remove o elemento do fim;
insert(i,T) -> adiciona um elemento T na posição i, empurrando os elementos;
clear() -> limpa todo o vector, deixando vazio.

Construtores

vector<int> v1;
-> É criado um vector vazio

vector<int> v2(3,5);
-> É criado um vector de tamanho inicial 3, cujas posições têm valor 5

vector<int> v3(v2.begin(),v2.end());
-> Os elementos de v2 são copiados a partir do iterador.

vector<int> v4(v3);
-> v4 copia o conteúdo de v3.

int vetor[] = { 4 , 8 , 90 , 15, 87 };
vector<int> v5(vetor,vetor + sizeof(vetor)/sizeof(int));
-> Vector copiando elementos de um vetor, usando o mesmo construtor de v3.

operator= e operator[]

v4 = v5;
-> v4 vira uma cópia de v5.

v5 = vector<int>();
-> Deste modo v5 se torna um vector vazio.

v3.operator=(v4);
-> É o mesmo que "v3 = v4"

cout << v3[3] << endl;
cout << v3.operator[](2) << endl;
-> As duas formas de acessar elementos do vector usando [].

Iteração

Este trecho de código imprime na tela os elementos de v3:

```
// Iteração normal
vector<int>::iterator it;
for(it = v3.begin() ; it < v3.end() ; it++) {
    cout << *it << " ";
}

// Iteração invertida
vector<int>:: reverse_iterator it2;
for(it2 = v3.rbegin() ; it2 < v3.rend() ; it2++) {
    cout << *it2 << " ";
}
```

-> Existem outros dois iteradores:
const_iterator e const_reverse_iterator.
Pesquise sobre eles.

Acesso aos elementos

```
cout << "v4 Front: "  << v4.front() << endl;
```
-> esta linha imprime na tela o primeiro elemento de v4.

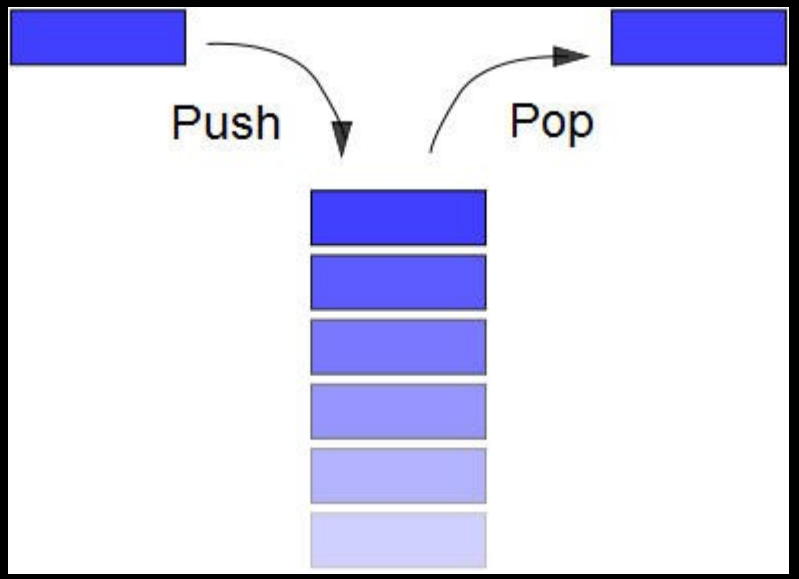
```
cout << "v4 Back : "  << v4.back()  << endl;
```
-> esta linha imprime na tela o último elemento de v4.

```
cout  << "v4 At 3: "  << v4.at(3)   << endl;
```
-> esta linha acessa o elemento de índice 3 de v4.

OBS: O operator[] funciona de modo similar ao at().

push_back() e pop_back()

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

int main (int argc, char **argv) {
   vector<int> v;
   // Elementos adicionados na seguinte ordem:
   // 1, 3, 9, 45, 33, 27
   v.push_back(1);
   v.push_back(3);
   v.push_back(9);
   v.push_back(45);
   v.push_back(33);
   v.push_back(27);

   while(!v.empty()) {
      cout << v.back() << " ";
      v.pop_back();
   }
   return 0;
}
```



Vector funcionando como uma pilha

insert() e size()

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

int main(int argc, char **argv) {
   vector<string> v;
   vector<string>::iterator it;
   cout << "v.size(): " << v.size() << endl;

   // Os elementos são inseridos sempre em v.begin(),
   // ou seja, na primeira posição do vector.
   v.insert(v.begin(),"A");
   v.insert(v.begin(),"B");
   v.insert(v.begin(),"C");
   v.insert(v.begin(),"D");

   // O "A" é inserido primeiro, mas ao fim, ele está
   // na quarta posição.
   for(it = v.begin() ; it < v.end() ; it++)  {
      cout << *it << " ";
   }
   cout << endl;
   cout << "v.size(): " << v.size() << endl;
   return 0;
}
```

- Note que o v.size() muda de valor após as inserções;
- O insert() pode ser escrito com outros parâmetros. Pesquise, se quiser.

resize() e capacity()

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

int main(int argc , char **argv)  {
   // Foi criado um vector com 5 elementos. Os 5 elementos são 14.1
   vector<double> v(5,14.1);
   v.push_back(3);

   // capacity() diz o espaço alocado para guardar elementos.
   // size() diz apenas o número de elementos guardados.
   cout << "v.capacity(): " << v.capacity() << endl;
   cout << "v.size(): " << v.size() << endl;

   // Resize apenas... modifica o tamanho.
   cout << "resize(10) e push_back(1)..." << endl;
   v.resize(10);
   v.push_back(1);

   cout << "v.capacity(): " << v.capacity() << endl;
   cout << "v.size(): " << v.size() << endl;

   for(unsigned int i = 0 ; i < v.size() ; i++)  {
      cout << v[i] << " ";
   }
   cout << endl;
   return 0;
}
```

Note que o espaço alocado, retornado pelo capacity(), dobra de tamanho sempre que ocorre um "overflow".

Um vector de tamanho inicial 1 dobraria a capacidade com os tamanhos: 2, 4, 8,16, etc.

Esta dobra frequente pode tornar o programa lento em alguns casos.

Alocar o espaço necessário desde o início com resize() é um modo de resolver este problema.

erase() e clear()

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

int main(int argc,char **argv) {
   // Vector criado com 6 elementos.
   vector<char> v;
   v.push_back('a'); v.push_back('b');
   v.push_back('c'); v.push_back('d');
   v.push_back('e'); v.push_back('f');

   v.erase(v.begin(), v.begin() + 2); // Os primeiros 2 elementos foram apagados.
   v.erase(v.end() - 1);              // O último elemento foi apagado.

   for(unsigned int i = 0 ; i < v.size() ; i++) {
      cout << v[i] << " ";
   }
   cout << endl;

   v.clear(); // Todos os elementos apagados.
   cout << "v.size(): " << v.size() << endl;

   return 0;
}
```

Observe que o erase() pode remover uma sequência de elementos ou apenas um.

O clear() remove todos.

deque<T>

"deque" significa "double-ended queue", "fila com dois fins". Na prática é como um vector, mas deque permite adição/remoção não só no fim, mas também no início, com os métodos push_front() e pop_front(). Assim como o vector, é diferente de uma lista encadeada comum por possuir o método at() e o operador []. Relembrando, listas encadeadas não possuem índices, mas vector e deque possuem.

Deque possui todos os métodos de vector citados no slide 15. Estes dois containers são diferentes apenas porque:
-> deque não possui os métodos capacity() e reserve(n);
-> deque possui os métodos push_front(T) e pop_front();

Deque: usado como um vector

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

int main(int argc, char **argv) {
    deque<double> d(3,0);

    deque<double>::const_iterator it;
    for(it = d.begin() ; it != d.end() ; it++) {
        cout << *it << " ";
    }
    cout << endl;

    d.clear();
    cout << "d.clear(), d.size(): " << d.size() << endl;
    return 0;
}
```

Deque como pilha:

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

int main (int argc, char **argv) {
    deque<char> d;
    d.push_back('a');
    d.push_back('b');
    d.push_back('c');
    d.push_back('d');

    while(!d.empty()) {
        cout << d.back() << " ";
        d.pop_back();
    }
    return 0;
}
```

Deque como fila:

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

int main (int argc, char **argv) {
    deque<char> d;
    d.push_back('a');
    d.push_back('b');
    d.push_back('c');
    d.push_back('d');

    while(!d.empty()) {
        cout << d.front() << " ";
        d.pop_front();
    }
    return 0;
}
```

Fim da apresentação! Mamma mia!

Leia também sobre set e map! L is real 2401!