STL – Standard Template Library

Prof. **João Paulo** R. R. Leite
joaopaulo@**unifei**.edu.br
ECOE44 & ECOE45 – Maratona de Programação

Existe um bom livro?

Sim, o **guia de consulta rápida** de Joel Saade é uma mão na roda, especialmente quando apenas queremos nos lembrar sobre funções e algoritmos, e suas assinaturas. Guia de referência, mas sem muita profundidade teórica.

Referência STL (C++ reference)

http://www.cplusplus.com/reference/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*** (ou gabaritos, em português). Criada por Alexander Stepanov e Meng Lee (Hewlett-Packard), adicionada ao C++ em 1994.

Mas o que são **templates** (gabaritos)?

São um recurso poderoso da linguagem C++, que possibilita especificar, **com um único segmento de código**, uma gama inteira de classes. Podemos, por exemplo, escrever um **único template de classe** para uma entidade genérica "pilha", e então fazer com que o C++ gere várias classes templates separadas, tais como pilha de **int**, de **float**, de **double**, de **string** e assim por diante, de acordo com a necessidade.

```cpp
#include <cstdio>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    vector<int> idades;
    vector<int>::iterator it;
    idades.push_back(20);
    idades.push_back(17);
    idades.push_back(21);

    printf("Vetor Desordenado: ");
    for(it = idades.begin(); it != idades.end(); ++it)
        printf("%d ", *it);
    sort(idades.begin(), idades.end());
    printf("\nVetor Ordenado: ");
    for(it = idades.begin(); it != idades.end(); ++it)
        printf("%d ", *it);
    return 0;
}
```

Repare que foi necessária a inclusão e duas bibliotecas: uma para utilização da classe **vector** e outra para a utilização do algoritmo de **ordenação**.

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

Estruturas de Dados

Uma estrutura de dados é uma forma de **armazenar e organizar dados**, provendo maneiras eficientes para a realização de inserções, consultas, buscas, atualizações e remoções.

Sabemos que as estruturas de dados em si não representam a solução de um problema. No entanto, conhecer as estruturas e utilizar a mais adequada em uma determinada situação pode ser a diferença entre acertar um problema e tomar uma penalização por limite de tempo excedido.

Como o objetivo desta disciplina é a programação competitiva, não focaremos nossos esforços na fundamentação teórica das estruturas de dados (ECO003/013) e na análise dos algoritmos de ordenação e busca (CCO005).

Nós precisamos apenas conhecer o maior número possível delas e – o mais importante de tudo – saber utilizá-las!

Precisamos conhecer:

- Pontos fortes e fracos
- Desempenho (complexidade)

Conhecer bem a STL irá **ajudar bastante** nas competições. A utilização de suas estruturas e algoritmos prontos nos ajuda a alcançar um nível mais alto produtividade.

Para começar, existem três conceitos que precisam ser conhecidos sobre a STL, para que possamos utilizá-la bem:

- Os **Containers** são as estruturas que armazenam valores de um tipo de dado (**int**, **float**, **string**, etc.) e encapsulam a estrutura de dados em si;
- Os **algoritmos** correspondem às ações a serem executadas sobre os containers (**ordenação**, **pesquisa**, p. ex.) e são utilizadas através de chamadas de funções;
- Os **iteradores** são componentes que percorrem os elementos de um container da mesma forma que um índice serve para percorrer os elementos de um *vetor* comum em estilo C.

Containers são classes implementadas com a finalidade principal de armazenar valores, que podem ser de tipos básicos ou tipos criados pelo usuário (classes ou structs).

Estão divididos em categorias:

- **Sequenciais:** vector, deque e list.
- **Adaptadores:** stack, queue e priority_queue.
- **Associativos Classificados:** set e map.

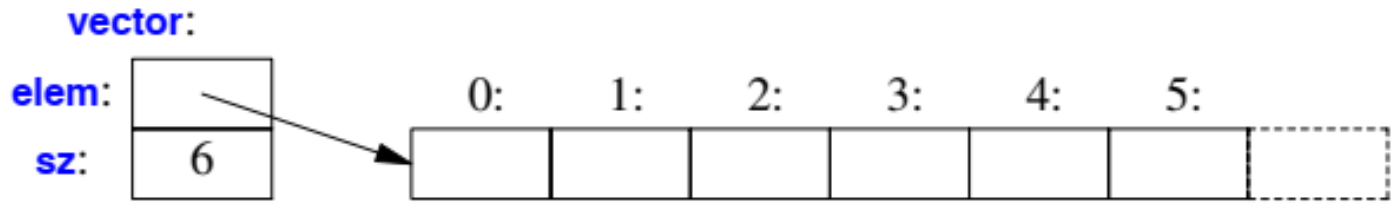
Dispõem de gerenciamento automático de memória, o que permite que o **tamanho do container varie dinamicamente**, aumentando ou diminuindo de acordo com a necessidade do programa.

Containers Sequenciais

- Nos **containers sequenciais**, os elementos estão em uma ordem linear na estrutura.
- Os containers sequenciais são:
  - **vector**, **deque** e **list**.

Container: vector <vector>

- Funciona como um vetor comum, estilo C, e podem ser manipulados com a mesma eficiência. No entanto, conseguem mudar seu tamanho dinamicamente e automaticamente.
- Os elementos são armazenados de forma contígua e podem ser acessados aleatoriamente através do operador []. No entanto, a maneira mais apropriada para percorrê-lo é através de um **iterador**.



Container: vector <vector>

- Suas principais operações são: **size()**, **push_back**(), **pop_back**(), **at**(), **operador []**, **empty**(), **clear**() e **swap**().

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

- O acesso a elementos aleatórios com operador [] é feito rapidamente, como em um vetor comum, devido à maneira como é alocado na memória.

- Exemplo de Código: **ex2**.cpp

Container: deque <deque>

- **Deque** (*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 em suas duas extremidades (*front* e *back*). No entanto, diferente dos vectors, seus elementos não estão alocados contiguamente na memória.

- Também **permitem o acesso direto** a todos os seus elementos, através do operador []. No entanto, pode não ser tão eficiente para acesso de elementos aleatórios, uma vez que não estão contíguos na memória (precisa de encadeamento).

Container: deque <deque>

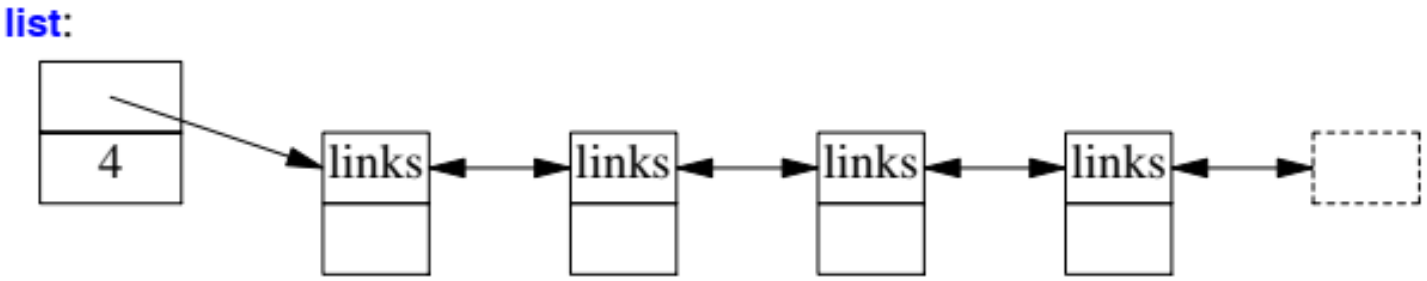
- Proveem funcionalidade similar aos vectors, mas com inserção e remoção eficiente de elementos também no início (front) da sequência, não somente no fim (back). Acabam sendo uma combinação entre vector e list, que não é tão boa quanto eles, sendo mais lento que um ou que outro para a maioria das aplicações. No entanto, caso seja sua necessidade específica, utilize sem medo.

- Suas principais operações são: **size()**, **push_back**(), **push_front()**, **pop_back()**, **pop_front()**, **front()**, **back()**, **at()**, **erase()**, **empty()**, **clear()**, **swap**(), **operador []**.

- Exemplo de Código: **ex3**.cpp

Container: list <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, exceto o primeiro e último.
- Não é possível utilizar o operador [].



Container: list <list>

- 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.

- Uma dica: Caso tudo o que queiramos seja uma sequência de elementos, devemos escolher um vector. A não ser que tenha uma boa razão, devemos sempre escolher vector, que tem melhor desempenho ao ser percorrido (find() e count()) e para ordenação e busca (sort() e binary_search()).

Container: list <list>

- Possui todas as operações já citadas. Possui uma operação **remove**, que remove elementos com um valor específico, e a operação **sort**, que ordena os valores da lista, entre outras.

- A maior desvantagem é que, comparadas a vector e deque, não possuem acesso direto aos elemento pela simples especificação de sua posição, seja utilizando a função **at()** ou o **operador []**.

- Para acessar, por exemplo, o décimo elemento da lista, é necessário iterar a partir de uma posição conhecida (como seu início ou fim) até aquela posição, o que leva um templo linearmente proporcional à distância entre eles.

- Exemplo de Código: **ex4**.cpp

Adaptadores de Container

- São classes que encapsulam um container específico e disponibilizam uma interface apenas com um conjunto de funções adaptadas para aquele tipo de estrutura que se deseja simular.

- Não permitem acesso aleatório a seus elementos, nem a utilização de iteradores. Portanto, não podem ser utilizados com os algoritmos da STL.

- Os adaptadores de container são:
  - **Pilha** (stack)
  - **Fila** (queue)
  - **Fila com Prioridade** (priority_queue)

Adaptadores: stack <stack>

- **É** um adaptador projetado especificamente para operar em um contexto **LIFO** (*last in, first out*), onde elementos são inseridos e removidos apenas do final (topo) do container.
- Trabalha como uma pilha da vida real.

- Operações básicas:
  - **empty()**
  - **size()**
  - **top()**
  - **push()**
  - **pop()**

Exemplo de Código: **ex5**.cpp

Adaptadores: queue <queue>

- **É** um adaptador projetado para operar em um contexto FIFO (first in, first out), onde elementos são inseridos no fim e removidos apenas do início do container.
- Funciona exatamente como uma fila da vida real.

- Operações básicas:
  - **empty**
  - **size**
  - **front**
  - **back**
  - **push** (*enqueue*, enfileira)
  - **pop** (*dequeue*, desenfileira)

Exemplo de Código: **ex5**.cpp

Algoritmos em Containers

- Existem, basicamente, duas operações que podem ser realizadas em estruturas sequenciais: **Ordenação** e **busca**.

1. Ordenação
   - Existe uma série de algoritmos de ordenação, que tomam o tempo $O(n^2)$. Precisam ser compreendidos, porém evitados.
     Ex.: bubble_sort, selection_sort, insertion_sort, etc.

   - Existem outros algoritmos, que possuem complexidade $O(n \log n)$. Estes devem ser utilizados quando necessário.
     Ex.: quick_sort, heap_sort, merge_sort.

   - Implementações em STL: **sort**, **partial_sort**, **stable_sort**.

Algoritmos em Containers

Não podem ser utilizados em lists, pois precisam de um iterador de acesso aleatório (list já tem um método de ordenação próprio).

- **sort:** Ordena os itens de uma sequência com complexidade O(n*logn). Pode também ordenar uma parte da sequência, mas leva em consideração apenas os elementos do intervalo.
- **partial_sort:** Ordena os itens de um intervalo de m elementos da sequência, levando em consideração todos os seus elementos. Leva O(n*logm).
- **stable_sort:** Ordena a lista mantendo as posições iniciais dos itens de mesmo valor. Leva tempo O(n*logn*logn) ou O(nlogn) se houver memória suficiente.

- Exemplo de Código: **ex6**.cpp

Algoritmos em Containers

- Busca em Sequências de Dados

  - **Busca linear**, O(n), quando buscamos um elemento percorrendo totalmente o vetor, do índice 0 a n-1.
  - Funções da STL: **find**(), **count**().
  - Veja: http://www.cplusplus.com/reference/algorithm/

  - **Busca Binária**, O(logn), implementada como lower_bound (ou binary_search) em STL. Necessita que a estrutura esteja ordenada: O(nlogn) no mínimo.
  - Exemplo de Código: **ex7**.cpp

Alguns Exercícios...

Lista de Exercícios 1 – URI Academic

Os exercícios devem ser resolvidos até 18/09, para **pontos extras** na primeira nota**.**