

Aula 11:

STL – Parte 1

Standard Template Library

ECOP13A - Programação Orientada a Objetos

Prof. André Bernardi

andrebernardi@unifei.edu.br

Universidade Federal de Itajubá



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** (como os que vimos na última aula). Criada por **Alexander Stepanov** e **Meng Lee** (Hewlett-Packard), adicionada ao C++ em 1994.

Relembrando... Mas o que são **templates**?

Um recurso da linguagem C++, que possibilita especificar, **com um único segmento de código**, uma gama inteira de classes relacionadas. Podemos, por exemplo, escrever um **único template** de classe para uma classe “**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.

STL

A STL é composta por um conjunto de componentes especificados pelo padrão ISO C++, e está presente em todos os compiladores da linguagem C++.

Recomenda-se fortemente sua utilização sempre que possível. Em geral, sua implementação é eficiente e livre de erros, além de ser um padrão internacional - o que garante facilidade na manutenção do seu código.

Seus componentes estão definidos no **namespace std** e são apresentados em um conjunto de arquivos de cabeçalho, que veremos no decorrer das aulas.

Alguns deles são: <vector>, <deque>, <list>, <map>, <set>, <queue>, <stack>, <algorithm>, <utility> e muitos outros.

Podemos enxergar a STL como uma **caixa de ferramentas** extra, que nos auxilia e traz soluções para muitos problemas de programação, principalmente envolvendo estruturas de dados.

Estruturas de Dados

A finalidade de uma estrutura é **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 quase **nunca 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 resolver um problema eficientemente ou não.

```

#include <stdio.h>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{
    // Cria estrutura de dados e iterador (para percorre-la)
    vector<int> idades;
    vector<int>::iterator it;

    // insere dados
    idades.push_back(20);
    idades.push_back(17);
    idades.push_back(21);

    // imprime desordenado
    printf("Vetor Desordenado: ");
    for(it = idades.begin(); it != idades.end(); ++it)
    {
        printf("%d ", *it);
    }

    // ordena vetor
    sort(idades.begin(), idades.end());
    printf("\nVetor Ordenado: ");
    for(it = idades.begin(); it != idades.end(); ++it)
    {
        printf("%d ", *it);
    }

    return 0;
}

```

Neste exemplo, utilizamos construções próprias da STL para criar um **contêiner para dados** do tipo **inteiro**, percorrer toda sua extensão através de “**ponteiros inteligentes**”, que chamaremos de **iteradores**, e aplicar nele um **algoritmo** de ordenação (já implementado na biblioteca: chamado **sort**).

Para isso, apenas incluímos duas bibliotecas: uma para utilização da classe **vector** e outra para a utilização do algoritmo de **ordenação**.

Existem **três** conceitos básicos importantes que precisam ser conhecidos sobre a **STL**, para que possamos utilizá-la bem:

- Os **Contêineres** são as estruturas que armazenam valores de um tipo de dado (**int**, **float**, **string**, um tipo definido pelo usuário, etc.) e encapsulam a estrutura de dados em si;
- Os **Algoritmos** correspondem às ações a serem executadas sobre os contêineres (**ordenação**, **pesquisa**, etc.) e são utilizados através de chamadas de funções;
- Os **Iteradores** são componentes que percorrem os elementos de um contêiner da mesma forma que um índice percorre os elementos de um *array* comum. Possuem algumas características em comum com ponteiros, mas seu comportamento é mais “inteligente”.

Contêineres são estruturas de dados implementadas na STL que servem para armazenar valores (de tipos básicos ou criados) e prover métodos de acesso, cada um segundo seu próprio tipo.

Estão divididos em algumas categorias:

Sequenciais: **vector**, **deque** e **list**.

Adaptadores: **stack**, **queue** e **priority_queue** (*heap*)

Associativos Classificados: **set** e **map** (*Balanced BST – Red-Black*)

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

Os contêineres sequenciais e associativos são chamados de **contêineres de primeira classe**.

Existem alguns outros tipos de dados que podem ser classificados como “**quase contêineres**”, dado que possuem muitos recursos em comum com os de primeira classe mas sem a mesma riqueza de recursos. São alguns deles:

- Arrays comuns, no estilo C;
- Strings de C++ (tipo **string**);
- **bitsets** (arrays de bits)

Contêineres Sequenciais

Nos **Contêineres sequenciais**, os elementos estão em uma **ordem linear** (ou unidimensional) na estrutura.

Os tipos podem ser **básicos** (int, float, etc.) ou **criados** pelo programador (structs, classes)

Os contêineres sequenciais são: **vector**, **deque** e **list**.



<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 **operator[]**.
- Sua utilização é importante especialmente quando não conhecemos o tamanho do vetor com antecedência.
- Suas principais operações são: **push_back()**, **pop_back()**, **size()**, **at()**, **operator[]**, **erase()**, **empty()**, **clear()** e **swap()**.
- Devemos utilizar um **iterador** para percorrê-lo.
- É necessário incluir o cabeçalho **<vector>**.
- São bastante eficientes no acesso aos elementos e na inserção e remoção de elementos no seu 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. **Consegue imaginar o porquê?**

Veja suas principais funções membro:

push_back(): acrescenta um elemento ao final do vetor.

pop_back(): remove o último elemento do vetor.

size(): retorna a quantidade de itens do vetor.

at() e **operator[]**: acessa um determinado elemento.

erase(): apaga elementos (apontados por um iterador)

empty(): testa se vetor está vazio.

clear(): elimina todo o conteúdo do vetor.

swap(): troca conteúdo do vetor que chama pelo de outro passado como parâmetro, e vice-versa.

Vejam um exemplo:

```

#include <vector>
#include <iostream>

using namespace std;

int main()
{
    vector<float> medidas;

    // Imprimimos o tamanho do vetor (inicial)
    cout << "Tamanho: " << medidas.size() << "\n";

    if(medidas.empty()) // verifica se está vazio
        cout << "O vetor esta vazio!\n\n";

    // Insere valores no final do vector
    medidas.push_back(15.6);
    medidas.push_back(23.6);
    medidas.push_back(2.9);
    medidas.push_back(17.3);
    medidas.push_back(11.9);
    medidas.push_back(7.7);

    // Imprimimos o tamanho do vetor (final)
    cout << "Tamanho: " << medidas.size() << "\n\n";

    // Podemos acessar itens individualmente (com operator[] ou at())
    cout << "Segundo item: " << medidas[1] << "\n";
    cout << "Quinto item: " << medidas.at(4) << "\n";

}

```

```

Tamanho: 0
O vetor esta vazio!

```

```

Tamanho: 6
Segundo item: 23.6
Quinto item: 11.9

```

<vector>

Foi falado que é necessário utilizar um “iterator” para percorrer o vector. Essa medida nos garante máximo desempenho e um código de acordo com os mais elevados padrões de programação em C++.

Mas o que é um iterator?

Iteradores possuem muitas características em comum com ponteiros e são utilizados para apontar elementos de contêineres de primeira classe. Eles armazenam informação sensível aos tipos específicos de contêineres para os quais apontam e, portanto, são implementados apropriadamente para cada tipo de contêiner, ainda que as suas operações mais importantes estejam presentes em qualquer iterator. Veja exemplos:

- O operador derreferenciador (*) derreferencia um iterator de modo que você possa obter o elemento para o qual ele aponta;
- A operação ++ em um iterator retorna um iterator que aponta para o próximo elemento do contêiner. A mesma coisa para aritmética simples como em iterator + 2, ou iterator--.

Lembre-se de que apenas contêineres de **primeira classe** podem ser manipulados através de iteradores: **vector**, **deque**, **list**, **set**, **multiset**, **map** e **multimap**.

Os contêineres de primeira classe da STL fornecem funções membro **begin()** e **end()**, que retornam iteradores apontando para o primeiro elemento do contêiner e para o primeiro elemento após o último elemento do contêiner, respectivamente.

Dica: Repare que a função **end()** não retorna um ponteiro para o último elemento, mas para um elemento inexistente que o seguiria na estrutura.

Portanto, através dos iteradores fica muito fácil percorrer um *vector* de ponta a ponta e realizar, por exemplo, a impressão de seus elementos. Veja o exemplo:

```
#include <vector>
#include <iostream>

using namespace std;

int main()
{
    vector<float> medidas;
    vector<float>::iterator it; // iterator próprio de vector<float>

    // Insere valores no final do vector
    medidas.push_back(15.6);
    medidas.push_back(23.6);
    medidas.push_back(2.9);
    medidas.push_back(17.3);
    medidas.push_back(11.9);
    medidas.push_back(7.7);

    // Imprime vetor com iterador
    cout << "Elementos do vector: ";
    for(it = medidas.begin(); it < medidas.end(); ++it)
        cout << *it << " ";
}
```

O iterador **it** recebe um valor inicial, apontando para o primeiro elemento de “medidas”. Seguimos incrementando seu valor até que aponte o último elemento.

<vector>

Voltando às funções membro de vector, vimos que poderíamos apagar valores através da função ***erase***, mas ela precisaria receber um iterador apontando para um determinado elemento a ser apagado.

Veja como ficaria:

```
// Podemos apagar itens (precisamos do iterador)
medidas.erase(medidas.begin()); // apaga primeiro item
medidas.erase(medidas.begin() + 2); // Apaga o terceiro item (primeiro já removido)
```

Agora, vejamos o **programa completo**:

arquivo vector1.cpp


```
#include <vector>
#include <iostream>

using namespace std;

int main()
{
    vector<float> medidas;
    vector<float>::iterator it;

    // Imprimimos o tamanho do vetor (inicial)
    cout << "Tamanho: " << medidas.size() << "\n";
    if(medidas.empty()) // verifica se está vazio
        cout << "O vetor esta vazio!\n\n";

    // Insere valores no final do vector
    medidas.push_back(15.6);
    medidas.push_back(23.6);
    medidas.push_back(2.9);
    medidas.push_back(17.3);
    medidas.push_back(11.9);
    medidas.push_back(7.7);
```

```
Tamanho: 0
O vetor esta vazio!

Tamanho: 6
Segundo item: 23.6
Quinto item: 11.9
Elementos do vector: 15.6 23.6 2.9 17.3 11.9 7.7
Vetor resultante: 23.6 2.9 11.9 7.7
```

```

// Imprimimos o tamanho do vetor (final)
cout << "Tamanho: " << medidas.size() << "\n";
// Podemos acessar itens individualmente (com operator[] ou at())
cout << "Segundo item: " << medidas[1] << "\n";
cout << "Quinto item: " << medidas.at(4) << "\n";

// Imprime vetor com iterador
cout << "Elementos do vetor: ";
for(it = medidas.begin(); it < medidas.end(); ++it)
    cout << *it << " ";
cout << endl;

// Podemos apagar itens (precisamos do iterador)
medidas.erase(medidas.begin()); // apaga primeiro item

// Apaga o terceiro item (pois o primeiro já foi removido)
medidas.erase(medidas.begin() + 2);

// Imprime vetor com iterador
cout << "Vetor resultante: ";
for(it = medidas.begin(); it < medidas.end(); ++it)
    cout << *it << " ";
cout << endl;
}

```

```

Tamanho: 0
0 vetor esta vazio!

Tamanho: 6
Segundo item: 23.6
Quinto item: 11.9
Elementos do vetor: 15.6 23.6 2.9 17.3 11.9 7.7
Vetor resultante: 23.6 2.9 11.9 7.7

```

<deque>

Deque (acrônimo de “*double-ended queue*”) é, como o próprio nome indica, uma fila com duas extremidades. É um tipo de contêiner 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.
- Proveem funcionalidade similar aos vectors, mas com inserção e remoção eficiente de elementos também no começo da sequência, e não somente no fim.
- Não é possível garantir que todos os seus elementos estarão alocados sequencialmente na memória.
- Suas principais operações são: **push_back()**, **push_front()**, **at()**, **erase()**, **empty()**, **clear()** e **swap()**, **operator[]**.
- É necessário incluir o cabeçalho **<deque>**.

Dica: Ainda que **operator[]** e **at()** realizem trabalho semelhante, há uma diferença. A função membro **at()** realiza uma checagem para verificar se o elemento está nos limites do contêiner. Caso não esteja, lança exceção **out_of_range**. O **operator[]** não realiza checagem e pode levar a erros em tempo de execução.

```

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

int main() {

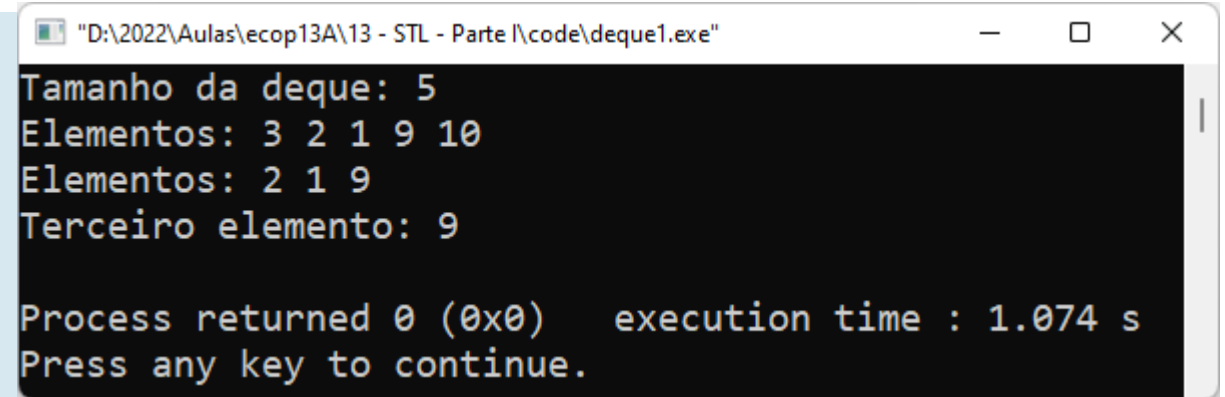
    deque<int> dados;

    // Podemos inserir elementos no inicio e fim da fila
    dados.push_front(1);
    dados.push_front(2);
    dados.push_front(3);
    dados.push_back(9);
    dados.push_back(10);

    cout << "Tamanho da deque: " << dados.size() << "\n";

    cout << "Elementos: ";
    for(deque<int>::iterator it = dados.begin(); it < dados.end(); ++it)
        cout << *it << " ";
    cout << endl;
}

```



```

"D:\2022\Aulas\ecop13A\13 - STL - Parte I\code\deque1.exe"
Tamanho da deque: 5
Elementos: 3 2 1 9 10
Elementos: 2 1 9
Terceiro elemento: 9

Process returned 0 (0x0)   execution time : 1.074 s
Press any key to continue.

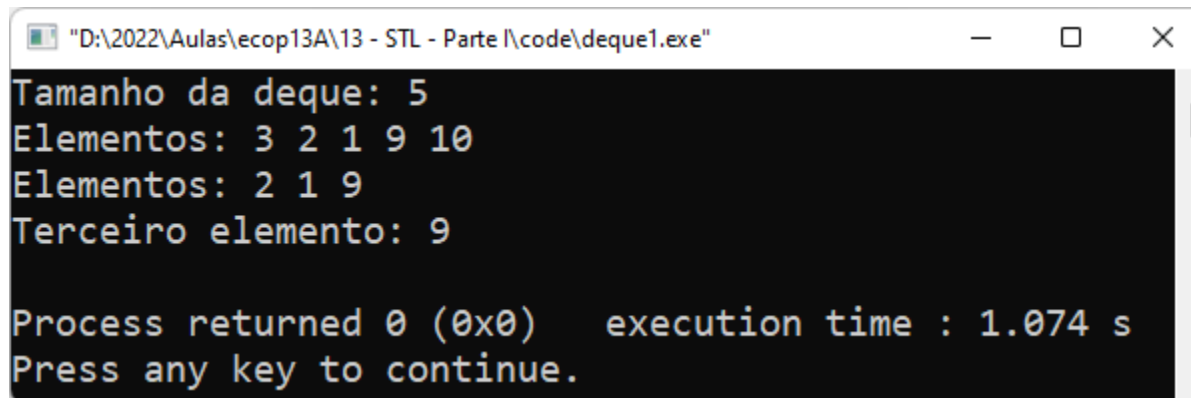
```

```
// também podemos remover elementos das duas extremidades
dados.pop_back();
dados.pop_front();

cout << "Elementos: ";
for(deque<int>::iterator it = dados.begin(); it < dados.end(); ++it)
    cout << *it << " ";
cout << endl;

cout << "Terceiro elemento: " << dados[2] << "\n";

return 0;
}
```



```
"D:\2022\Aulas\ecop13A\13 - STL - Parte I\code\deque1.exe"
Tamanho da deque: 5
Elementos: 3 2 1 9 10
Elementos: 2 1 9
Terceiro elemento: 9

Process returned 0 (0x0)   execution time : 1.074 s
Press any key to continue.
```

<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 contêiner 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.
- Possui todas as operações já citadas. Possui uma operação **remove**, que remove elementos com um valor específico, e **insert**, que insere um novo elemento antes de um elemento apontado pelo iterador.
- É necessário incluir o cabeçalho **<list>**

```

#include <iostream>
#include <list>
#include <string>

using namespace std;

int main()
{
    list<string> nomes;
    list<string>::iterator it;

    // Inserindo dados(back, front, random)
    nomes.push_back("Joao");
    nomes.push_back("Paulo");
    nomes.push_front("Andre");

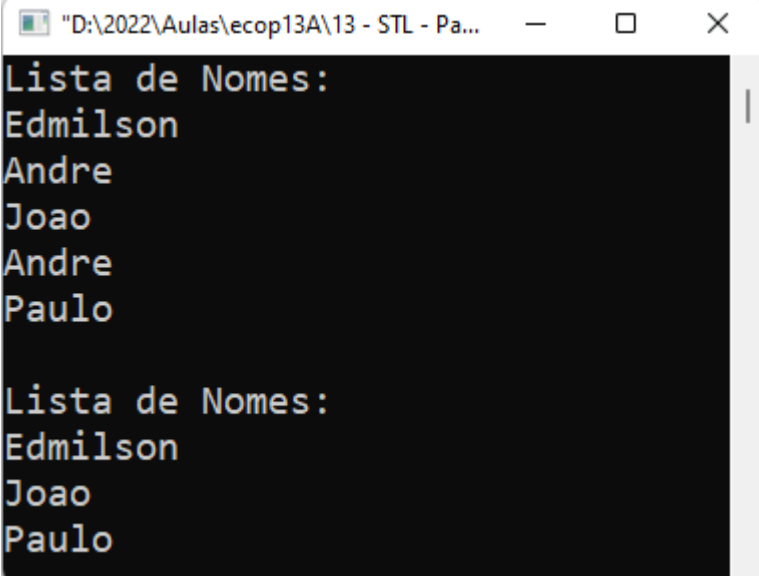
    it = nomes.begin();
    nomes.insert(it, "Edmilson");
    ++it; ++it;
    nomes.insert(it, "Andre");

    cout << "Lista de Nomes: " << endl;
    for(it = nomes.begin(); it != nomes.end(); ++it)
        cout << *it << endl;

    // Removendo todos os Andres
    nomes.remove("Andre");
    cout << "\nLista de Nomes: " << endl;
    for(it = nomes.begin(); it != nomes.end(); ++it)
        cout << *it << endl;
}

```

A função insert, assim como erase, recebe um iterator que aponta para um determinado elemento da lista. O elemento a ser inserido, entra antes dele no encadeamento de elementos.



```

D:\2022\Aulas\ecop13A\13 - STL - Pa...
Lista de Nomes:
Edmilson
Andre
Joao
Andre
Paulo

Lista de Nomes:
Edmilson
Joao
Paulo

```

Adaptadores de Contêiner

Adaptadores de Contêiner são classes que **encapsulam um contêiner** específico e nos proveem como interface pública apenas um conjunto de funções membro, que servem para aquele tipo de estrutura que se quer simular, com suas regras específicas.

Não suportam uso de iteradores e, portanto, não podem ser utilizados com os algoritmos da STL. Também não permitem acesso aleatório a seus elementos.

Os adaptadores de contêiner são:

- **Pilha** (stack)
- **Fila** (queue)
- **Fila com Prioridade** (priority_queue) – **Próxima Aula**

Adaptador de Contêiner - **stack**

stack é um adaptador de contêiner projetado especificamente para operar em um contexto **LIFO** (*last in, first out*), onde elementos são SEMPRE inseridos e removidos apenas do final (topo) do contêiner.

- Trabalha como uma pilha da vida real.
 - Último a entrar, primeiro a sair.

Cabeçalho: **<stack>**

Operações básicas:

- **empty**
- **size**
- **top**
- **push** (trabalha como `push_back`)
- **pop** (trabalha como `pop_back`)



Adaptador de Contêiner - **queue**

queue é um adaptador de contêiner 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 contêiner.

- Funciona exatamente como uma fila da vida real.
 - Primeiro a entrar, primeiro a sair.

Cabeçalho: **<queue>**

Operações básicas:

- **empty**
- **size**
- **front**
- **back**
- **push** (é um `push_back`)
- **pop** (é um `pop_front`)



```

#include <iostream>
#include <stack>
#include <queue>

using namespace std;

int main()
{
    stack<int> pilha;
    queue<int> fila;

    // insere na pilha e na fila
    for(int i = 0; i < 5; i++) {
        pilha.push(i);
        fila.push(i);
    }

    // imprime pilha
    cout << "Pilha: ";
    while(!pilha.empty()) {
        cout << pilha.top() << " ";
        pilha.pop();
    }
    cout << endl;

    // imprime fila
    cout << "Fila: ";
    while(!fila.empty()) {
        cout << fila.front() << " ";
        fila.pop();
    }
    cout << endl;
}

```

```

Pilha: 4 3 2 1 0
Fila: 0 1 2 3 4

```

Adaptadores **não são** contêineres de primeira classe, uma vez que não implementam realmente uma estrutura de dados e não suportam iteradores. Para elas, o próprio programador pode escolher a estrutura que mais combina com o adaptador. Veja:

A implementação padrão da **stack** é realizada adaptando-se uma **deque**. Para criar pilhas adaptadas de **vector** e **list**, devemos declarar:

```

stack<int, vector<int>> vec_stack;
stack<int, list<int>> vec_list;

```

Para **queues**, a melhor adaptação é realizada com a **deque**. Não é possível adaptar um **vector** como queue (não possui uma função **pop_front()**, pois somente remove itens do final do vector). **list** continua sendo uma opção.

Dicas de Stroustrup:

1. Não reinvente a roda! Utilize as estruturas da **biblioteca padrão**.
2. Quando tiver escolha, **prefira a biblioteca padrão a bibliotecas de terceiros**, mas não pense que ela resolve todos os problemas.
3. Não se esqueça de incluir os **arquivos de cabeçalho**.
4. Não se esqueça que tudo está no **namespace std**.
5. Utilize **vector** como sua estrutura de dados padrão de C++. Prefira `vector<T>` a `T[]`.
6. Prefira estruturas de dados mais compactas e contíguas.
7. **vectors** são alocados de maneira contígua. **lists** são listas encadeadas.
8. Prefira o uso de **push_back** para inserir dados em um contêiner.
9. Operações de **push_back** normalmente são mais rápidas em vector que em list.
10. Não assuma que `[]` faz checagens de limite. Use **at()** para tal.
11. Os **adaptadores de contêiner** não proveem acesso direto à estrutura que estão adaptando.
12. **Conheça bem** os contêineres e suas vantagens.



Referências

- <https://cplusplus.com/reference/>
 - <https://cplusplus.com/reference/vector/vector/>
 - <https://cplusplus.com/reference/deque/deque/>
 - <https://cplusplus.com/reference/list/list/>
 - <https://cplusplus.com/reference/stack/stack/>
 - <https://cplusplus.com/reference/queue/queue/>
- Material de aula ECOPo3 - Prof. João Paulo R. R. Leite - 2021