STL – Standard Template Library

- A STL C++ é uma biblioteca padronizada de funções, fortemente baseada no uso de templates, que oferece um conjunto de classes de uso genérico, contendo:
  - contêineres (estruturas de dados, como vetores, pilhas, listas e filas);
  - iteradores (objetos que percorrem elementos de um conjunto); e
  - algoritmos básicos (principalmente os destinados a busca e classificação).
- Uma das principais vantagens do uso desta biblioteca está na simplificação do trabalho com estruturas de dados, uma vez que o código baseado em ponteiros é complexo.

Iterator

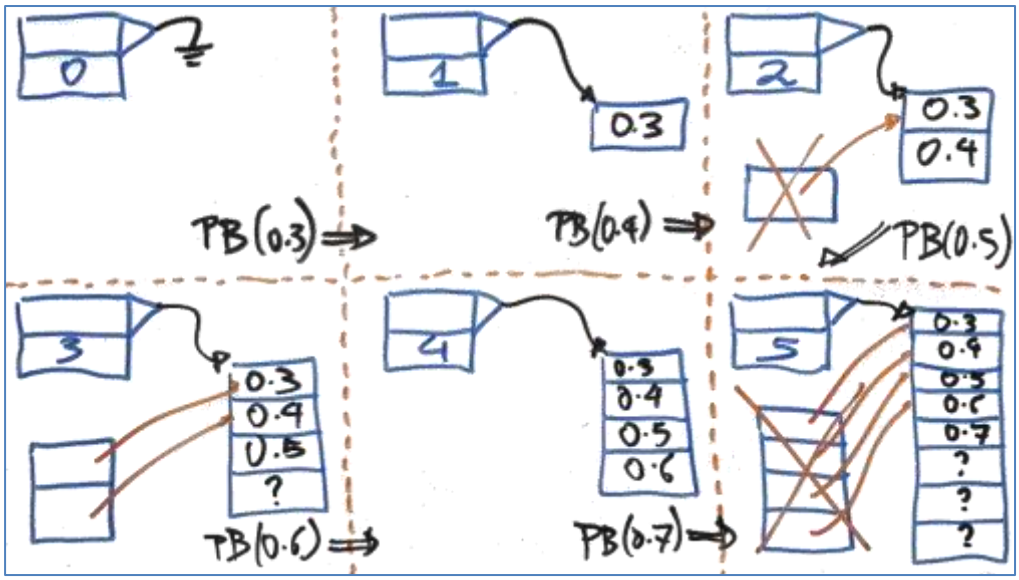
- Os iteradores, similares a ponteiros, são usados para apontar para os elementos dos contêineres. Cada iterador é específico para um tipo do contêiner a percorrer.
- Contêineres oferecem os métodos `begin()` e `end()` para o trabalho com iteradores.
  - `begin()` – retorna iterador que aponta para o primeiro elemento do contêiner.
  - `end()` – retorna iterador que aponta para depois do último elemento do contêiner.
- O operador `*` é usado para acessar o elemento apontado (similar à desreferenciação em ponteiros)

```
// Cria e aloca contêiner V de 6 objetos do tipo ”tipo_do_objeto”
contêiner<tipo_do_objeto> V(6);
// cria um iterador 'var' para objetos ”tipo_do_objeto”
contêiner<tipo_do_objeto>::iterator var;

// percorre o conteiner
for (var = V.begin(); var != V.end(); var++ ) {
  cout << "Objeto armazenado no vetor: " << *var << endl;
}
```

Vector (vetor)

- Vector é um tipo de contêiner baseado em arrays. Porém seu tamanho pode mudar dinamicamente, com o armazenamento sendo tratado automaticamente pelo contêiner.
- Como esta estrutura de dados trabalha com posições de memória contíguas, o acesso direto a seus elementos também pode ser feito através do operador subscrito `[ ]` e não exclusivamente através de iteradores.
- Internamente, os vectors usam um array alocado dinamicamente para armazenar seus elementos. Esse array pode precisar ser realocada para aumentar de tamanho quando novos elementos são inseridos, o que implica alocar uma nova área de memória e mover todos os elementos para ela. Essa é uma tarefa custosa em tempo de processamento e, portanto, os arrays não são realocados cada vez que um elemento é adicionado ao contêiner.
  - Na maioria dos compiladores, o array dobra de tamanho sempre que for preciso adicionar um novo elemento para o qual a memória alocada atual não seja suficiente.
  - Comparados com arrays, os vectors consomem mais memória em troca da capacidade de gerenciar armazenamento e crescer dinamicamente de maneira mais eficiente.
  - Exemplo: se o vector tem 4 elementos, o array está alocado com capacidade para armazenar exatamente os 4 elementos e é preciso armazenar mais um (o 5º elemento), o array será realocado para ter capacidade de armazenar 8 (2 x 4) elementos, dos quais 5 serão utilizados imediatamente e 3 ficam de reserva para eventual uso futuro.



- Os vetores são muito eficientes acessando seus elementos e adicionando ou removendo elementos do final. Para operações que envolvem a inserção ou remoção de elementos em posições diferentes da extremidade, eles não apresentam bom desempenho.
- Os métodos frequentemente utilizados com vectors são:
  - `size`() – retorna o tamanho (número de elementos do contêiner).
  - `empty`() – testa se o contêiner está vazio.
  - `operator[]()` – acesso ao elemento (sem checagem do índice).
  - `at()` – acesso ao elemento (com checagem do índice).
  - `front()` – acesso ao primeiro elemento.
  - `back()` – acesso ao último elemento.
  - `push_back`(elemento) – insere novo elemento no fim do contêiner.
  - `pop_back`() – exclui último elemento do contêiner.
  - `clear`() – limpa o contêiner (apaga todos os elementos).
  - `capacity`() – exclusivo do `vector`. Retorna o tamanho da atual memória alocada (quantos elementos pode conter sem necessidade de realocar).
- Outros métodos que podem ser utilizados com vectors, mas que não são tão eficientes pois envolvem necessidade de realocação de memória e/ou cópia dos elementos, são:
  - `insert`(posição, elemento) – insere novo elemento antes da posição (iterador).
  - `erase`(posição) – exclui elemento na posição (iterador) especificada.
  - `erase`(posição inicial, posição final+1) – exclui elementos da faixa do iterador inicial até antes do iterador final.

Algoritmos STL

- A STL define algoritmos padrão que manipulam dados armazenados em contêineres.
- Os algoritmos podem ser utilizados em quase todos os contêineres e, em alguns casos, até em arrays (que não são contêineres STL).
- Os algoritmos são funções e não métodos das classes (com raras exceções: `sort` de listas).
- Exemplos de algoritmos:
  - `for_each`(posição inicial, posição final+1,função) – executa a função dada para cada elemento dentro da faixa, passando-o como parâmetro de chamada da função.
  - `find`(posição inicial, posição final+1,valor) – procura dentro da faixa um elemento com o valor dado (utiliza o operador ==).
  - `find_if`(posição inicial, posição final+1,função) – procura dentro da faixa um elemento para o qual a função dada retorne `true`.

- `count`(posição inicial, posição final+1,valor) – conta o número de ocorrências dentro da faixa de elementos com o valor dado (utiliza o operador ==).
- `count_if`(posição inicial, posição final+1,função) – conta o número de ocorrências dentro da faixa de elementos para o qual a função dada retorne `true`.
- `reverse`(posição inicial, posição final+1) – reverte a ordem dos elementos na faixa.
- `sort`(posição inicial, posição final+1) – ordena os elementos dentro da faixa (utiliza o operador <). Não deve ser utilizado com contêineres do tipo list.
- `sort`(posição inicial, posição final+1,função) – ordena os elementos dentro da faixa, utilizando a função dada como comparador (se função(A,B) retorna true, então A deve vir antes de B no contêiner ordenado) . Não deve ser utilizado com lists.

- Os algoritmos STL que recebem como parâmetro uma função (`for_each`, `find_if`, `count_if`, etc.) aceitam qualquer elemento ao qual possam ser aplicados parênteses, já que se trata de programação genérica. Por exemplo, pode ser um objeto de uma classe para o qual esteja sobrecarregado o `operator()`.

```
// Define funcao equal45
inline bool equal45(int i)
{ return (i==45); }
```

```
// Define classe Eq45
class Eq45 {
public:
  inline bool operator()(int i)
  { return (i==45); }
};
// Cria objeto equal45 da classe Eq45
Eq45 equal45;
```

```
vector<int> X;
vector<int>::iterator pX;
…
// Retorna primeiro inteiro do vetor com valor igual a 45.
// Funciona com equal45 definido de qualquer uma
// das 2 maneiras acima - funcao ou objeto
pX = find_if(X.begin(), X.end(), equal45);
```

  - Esse tipo de objeto que se assemelha a uma função (porque pode ser chamado com parênteses) é denominado objeto-função ou *functor*.
  - O *functor* é a alternativa para se utilizar os algoritmos STL que recebem como parâmetro uma função nas situações onde seria necessário um parâmetro adicional para a função. Nesses casos, o *functor* pode armazenar esse valor como dado membro.

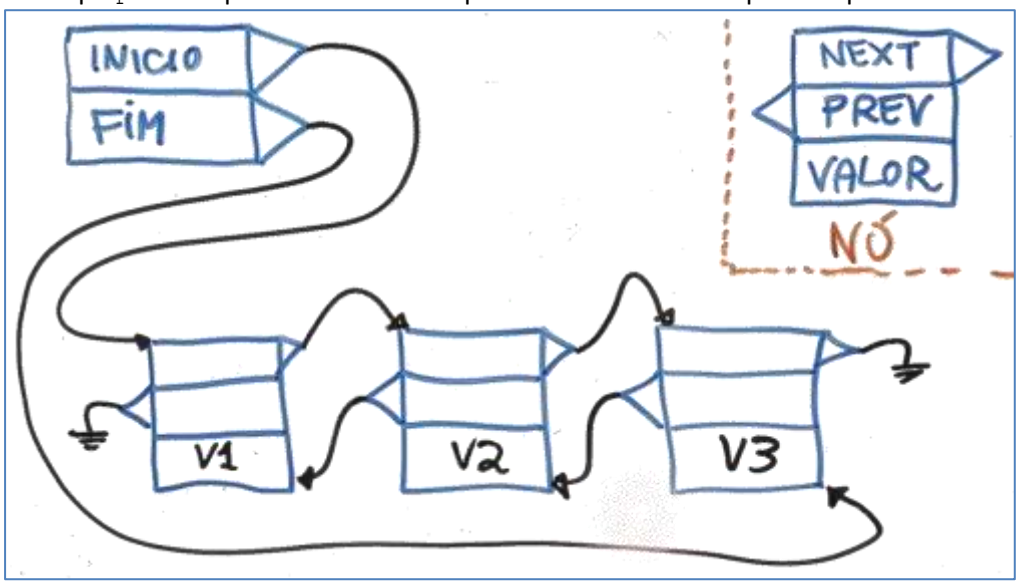
```
// Define classe Eq
class Eq {
private:
  int valor;
public:
  // Construtor
  inline Eq(int V):
    valor(V) {}
  inline bool operator()(int i)
  { return (i==valor); }
};
```

```
vector<int> X;
vector<int>::iterator pX;
int N;
…
// Retorna 1° inteiro do vetor
// com valor igual a N.
Eq equalN(N);
pX=find_if(X.begin(), X.end(),
           equalN);
// ou simplesmente
pX=find_if(X.begin(), X.end(),
           Eq(N));
```

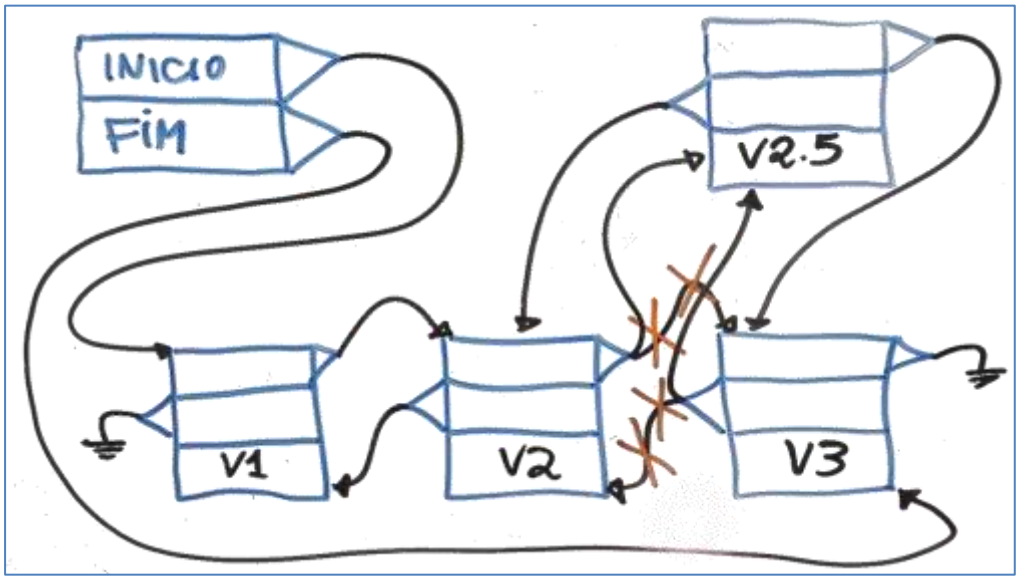
List (lista)

- List é um tipo de contêiner sequencial que trabalha com operações de inserção e exclusão de elementos em qualquer posição do contêiner.
- Suporta iteradores de acesso bidirecional, o que permite percorrer nos dois sentidos.

- O operador subscrito `[ ]` não é suportado pela list, que só permite acesso sequencial.
- É implementada como uma lista duplamente encadeada, que é uma estrutura de dados que consiste de um conjunto de *nós* sequencialmente ligados.
  - Dois ponteiros permanentes da lista apontam para o primeiro e último nós.
  - Além do campo que armazena o dado propriamente dito, cada nó contem dois campos adicionais, chamados de links ou enlaces, que são ponteiros para o nó anterior (`prev`) e para o nó posterior (`next`) na sequência de nós.
  - O campo `prev` do primeiro nó e o campo `next` do último nó apontam para `NULL`.



  - A inserção de um novo elemento em qualquer posição da lista é feita em tempo constante, independentemente da posição ou do tamanho atual da lista, pois requer apenas a alocação de memória para conter o novo nó e a alteração dos valores de 4 ponteiros: `prev` e `next` do novo nó (apontando para o predecessor e o sucessor dele, respectivamente), do `next` do predecessor e do `prev` do sucessor. Nenhum dos elementos já presentes na lista tem sua posição de memória alterada.

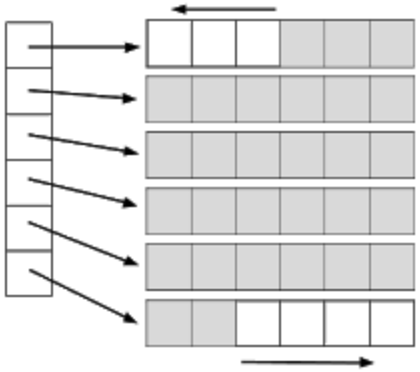


- Comparadas a vetores, as listas têm um desempenho geralmente melhor na inserção, extração e movimentação de elementos em qualquer posição dentro do contêiner para o qual um iterador já foi obtido e, portanto, também em algoritmos que fazem uso intensivo dessas operações, como algoritmos de ordenamento.

- A principal desvantagem de listas em comparação com vetores é que elas não têm acesso direto aos elementos por sua posição. Para acessar um elemento em uma lista é preciso iterar do início ou do fim da lista até a posição do elemento, o que leva tempo linearmente dependente da distância entre eles. Elas também consomem alguma memória extra para manter as informações de vinculação associadas a cada nó, o que pode ser um fator importante para grandes listas de elementos de tamanho pequeno.
- Os métodos frequentemente utilizados com listas são:
  - `size`() – retorna o tamanho (número de elementos do contêiner).
  - `empty`() – testa se o contêiner está vazio.
  - `front()` – retorna o primeiro elemento do contêiner (não remove).
  - `back()` – retorna o último elemento do contêiner (não remove).
  - `push_back`(elemento) – insere novo elemento no fim do contêiner.
  - `push_front`(elemento) – insere novo elemento no início do contêiner.
  - `pop_back`() – exclui último elemento do contêiner.
  - `pop_front`() – exclui primeiro elemento do contêiner.
  - `clear`() – limpa o contêiner (apaga todos os elementos).
  - `insert`(posição, elemento) – insere novo elemento antes da posição (iterator).
  - `erase`(posição) – exclui elemento na posição (iterator) especificada.
  - `erase`(posição inicial, posição final+1) – exclui elementos da faixa especificada.
  - `remove`(valor) – remove elementos com o valor dado (utiliza o operador ==).
  - `unique`() – remove elementos duplicados consecutivos (utiliza o operador ==).
  - `unique`(funcão) – remove elementos consecutivos para os quais função(A,B) é true.
  - `sort`() – para listas, o método `sort` deve ser utilizado ao invés do algoritmo genérico `sort` do STL. Ordena toda a lista usando o operador <.
  - `sort`(funcão) –Ordena toda a lista usando a função dada como comparador (se função(A,B) retorna true, então A deve vir antes de B na lista ordenada).

Deque

- Double-Ended QUEue = fila com duas extremidades.
- Permite acesso direto aos elementos através do operador subscrito `[ ]`.
- Fornece uma funcionalidade semelhante ao `vector`, mas com inserção e exclusão de elementos eficientes também no início da sequência, e não apenas no final.
- Ao contrário do `vector`, não é garantido que o deque armazene todos os seus elementos em locais de armazenamento contíguos (um após o outro).
- Pode ser implementado como uma lista de blocos de tamanho fixo, sendo que apenas o primeiro e o último blocos podem estar parcialmente preenchidos, com novos blocos podendo ser acrescentados nas extremidades, aumentando a lista.



- São mais complexos que vetores, mas podem ser mais eficientes em certas circunstâncias, especialmente com sequências muito longas, em que as realocações se tornam mais caras.

- Para inserções ou remoções de elementos em posições diferentes do início ou do fim, os deques apresentam um desempenho pior do que as listas, porém melhor do que os vetores, já que eles podem escolher o melhor sentido para reposicionar os elementos.
- Os métodos frequentemente utilizados são os mesmos do `vector`, sem o `capacity` e incluindo além deles:
  - `push_front`(elemento) – insere novo elemento no início do contêiner.
  - `pop_front`() – exclui primeiro elemento do contêiner.

Set (conjunto)

- Contêiner especificamente projetado de tal forma que seus elementos são únicos (sem nenhuma repetição) e ordenados (usando `operator<`).
- Os elementos em um conjunto não podem ser modificados uma vez que estejam no contêiner, mas podem ser inseridos ou removidos.
- Conjuntos são tipicamente implementados como "árvores binárias de busca".
- Não suporta o operador subscrito `[ ]`.
- Possui os métodos:
  - `insert(`elemento`)` – insere novo elemento no local adequado do contêiner. Não duplica o elemento caso já faça parte do conjunto.
  - `erase`(posição) – exclui elemento na posição (iterator) especificada.
  - `lower_bound(`valor`)` – retorna iterator para o primeiro elemento menor ou igual que valor.
  - `upper_bound(`valor) – retorna iterator para o primeiro elemento maior que valor.

Stack (pilha)

- Projetada especificamente para operar em um contexto LIFO (last-in first-out), onde os elementos são inseridos e extraídos apenas em uma extremidade do contêiner.
- Inserções e remoções são permitidas apenas no fim do contêiner.
- Não suporta o operador subscrito `[ ]` nem permite o percurso do contêiner.
- Implementada como uma adaptação de outro contêiner (`vector`, `deque` ou `list`).
- Possui os métodos:
  - `top()` – retorna o elemento do topo do contêiner (não remove).
  - `push`(elemento) – insere novo elemento no topo do contêiner.
  - `pop`() – exclui elemento do topo do contêiner.

Queue (fila)

- Projetada especificamente para operar em um contexto FIFO (first-in first-out), onde os elementos são inseridos em uma extremidade do contêiner (no final) e extraídos na outra.
- Inserções e remoções são permitidas apenas na extremidade correta do contêiner.
- Não suporta o operador subscrito `[ ]` nem permite o percurso do contêiner.
- Implementada como uma adaptação de outro contêiner (`deque` ou `list`).
- Possui os métodos:
  - `front()` – retorna o primeiro elemento do contêiner (não remove).
  - `back()` – retorna o último elemento do contêiner (não remove).
  - `push`(elemento) – insere novo elemento no fim do contêiner.
  - `pop`() – exclui elemento do início do contêiner.

Priority_queue (fila de prioridades)

- Contêiner especificamente projetado de tal forma que seu elemento do topo é sempre o maior dos elementos (usando `operator<`).
- Semelhante a um "*heap*", no qual os elementos podem ser inseridos a qualquer momento e apenas o elemento máximo pode ser recuperado (aquele no topo da fila de prioridades).
- Não suporta o operador subscrito `[ ]` nem permite o percurso do contêiner.
- Implementada como uma adaptação de outro contêiner (`vector` ou `deque`).
- Utiliza os algoritmos `make_heap()`, `push_heap()` e `pop_heap()`, quando necessário.
- Possui os métodos:
  - `top()` – retorna o elemento máximo do contêiner (não remove).
  - `push`(elemento) – insere novo elemento no local adequado do contêiner.
  - `pop`() – exclui elemento do topo do contêiner (o elemento máximo).

Map

- O map associa uma chave a um item. Para definir um objeto do tipo map, é preciso especificar 2 tipos, o da chave e o dos itens.
- Por exemplo, um vector<string> é como um map com chaves tipo int e items tipo string.
- Cada elemento de um map é um par (`pair`). O campo `first` acessa a chave e o campo `second` acesa o item.
- Os algoritmos de busca (`find`) fazem a procura pela chave.

Comparativo

OPERAÇÃO	**`vector`**	**`deque`**	**`list`**	**`stack`**	**`queue`**	**`priority queue`**	**`set`**
Tem operador subscrito `[ ]`	SIM	SIM	NÃO	NÃO	NÃO	NÃO	NÃO
Permite inserção no fim	SIM	SIM	SIM	SIM	SIM	NÃO	NÃO
Permite remoção do fim	SIM	SIM	SIM	SIM	NÃO	SIM	NÃO
Permite inserção no início	NÃO	SIM	SIM	NÃO	NÃO	NÃO	NÃO
Permite remoção do início	NÃO	SIM	SIM	NÃO	SIM	NÃO	NÃO
Permite inserção/remoção do meio	RUIM	RUIM	SIM	NÃO	NÃO	NÃO	NÃO
Permite inserção por valor	RUIM	RUIM	SIM	NÃO	NÃO	SIM	SIM
Permite remoção por valor	RUIM	RUIM	SIM	NÃO	NÃO	NÃO	SIM
Permite percurso e acesso aos elementos internos	SIM	SIM	SIM	NÃO	NÃO	NÃO	SIM
Permite modificação de valor de elemento	SIM	SIM	SIM	NÃO	NÃO	NÃO	NÃO
Permite elementos repetidos	SIM	SIM	SIM	SIM	SIM	SIM	NÃO
Armazena elementos consecutivamente em memória	SIM	NÃO	NÃO	NÃO	NÃO	NÃO	NÃO
Mantém elementos em ordem temporal	NÃO	NÃO	NÃO	SIM	SIM	NÃO	NÃO
Mantém elementos em ordem de valor	NÃO	NÃO	NÃO	NÃO	NÃO	SIM	SIM