

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.
- Devem ser implementados exatamente com o mesmo tipo do contêiner a percorrer.
- Contêineres oferecem os métodos `begin()` e `end()` para o trabalho com iteradores.
- O operador `*` é usado para acessar o elemento apontado.

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

Vector (vetor)

- Vector é um tipo de contêiner sequencial, baseado em arrays.
- Como esta estrutura de dados trabalha com posições de memória contíguas, o acesso direto a seus elementos também pode ser feito através do operador subscripto `[]`.
- Os métodos frequentemente utilizados são:
 - `begin()` – retorna iterador para o primeiro elemento do contêiner.
 - `end()` – retorna iterador para depois do último elemento do contêiner.
 - `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).
 - `empty()` – testa se o contêiner está vazio.
 - `size()` – retorna o tamanho (número de elementos do contêiner).
 - `capacity()` – exclusivo do `vector`. Retorna o tamanho da atual memória alocada (quantos elementos pode conter sem necessidade de realocar).

Exemplo utilizando o operador subscripto `[]` de acesso não-sequencial:

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

using namespace std;
```

```

int main()
{
    vector<int> meuVetor; // cria um vetor de inteiros vazio

    if (meuVetor.empty()) // testa se o vetor está vazio
        cout << "Vetor vazio!" << endl;
    else
        cout << "Vetor com elementos!" << endl;

    meuVetor.push_back(7); // inclui elemento no fim do vetor
    meuVetor.push_back(11);
    meuVetor.push_back(2006);

    // vai imprimir tres elementos {7, 11, 2006}
    cout << "Imprimindo o vetor...: ";
    for (int i = 0; i < meuVetor.size(); i++)
        cout << meuVetor[i] << ' ';
    cout << endl;

    meuVetor.pop_back(); // retira o último elemento

    // agora, soh vai imprimir dois {7, 11}
    cout << "Meu vetor, de novo...: ";
    for (int i = 0; i < meuVetor.size(); i++)
        cout << meuVetor[i] << ' ';
    cout << endl;

    return 0;
}

```

Exemplo utilizando iteradores (acesso sequencial):

```

#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> meuVetor; // cria um vetor de inteiros vazio
    vector<int>::iterator j; // cria um iterador de inteiros

    meuVetor.push_back(7); // inclui no fim do vetor um elemento
    meuVetor.push_back(11);
    meuVetor.push_back(2006);

    // vai imprimir três elementos {7, 11, 2006}
    cout << "Imprimindo o vetor...: ";
    for ( j = meuVetor.begin(); j != meuVetor.end(); j++ )
        cout << *j << ' ';
    cout << endl;

    // insere 55 como segundo elemento, deslocando os demais
    meuVetor.insert( meuVetor.begin() + 1, 55);

    // agora, vai imprimir quatro elementos {7, 55, 11, 2006}
    cout << "Inseri no meio do vetor...: ";
    for ( j = meuVetor.begin(); j != meuVetor.end(); j++ )
        cout << *j << ' ';
    cout << endl;
}

```

```

// retira 11 da lista (terceira posição)
meuVetor.erase( meuVetor.begin() + 2);

// agora, tem que imprimir três de novo {7, 55, 2006}
cout << "Retirei no meio do vetor..: ";
for ( j = meuVetor.begin(); j != meuVetor.end(); j++ )
    cout << *j << ' ';
cout << endl;

meuVetor.clear(); // limpa todo o vetor

return 0;
}

```

Exemplo utilizando classe criada pelo programador:

```

#include <iostream>
#include <vector>

using namespace std;

class Pessoa
{
private:
    string nome;
    int idade;
public:
    inline Pessoa(string no, int id): nome(no), idade(id) {}
    inline string getNome() const {return nome;}
    inline int getIdade() const {return idade;}
};

int main()
{
    vector<Pessoa> VP;
    vector<Pessoa>::iterator ptr;

    VP.push_back(Pessoa("Joao", 25));
    VP.push_back(Pessoa("Maria", 32));
    VP.push_back(Pessoa("Carla", 4));
    VP.push_back(Pessoa("Abel", 30));

    // percorrendo a lista com indices
    for(int i = 0; i < VP.size(); i++)
    {
        cout << "Nome: " << VP[i].getNome();
        cout << " - Idade: " << VP[i].getIdade() << endl;
    }
}

```

Algoritmos STL

- A STL define vários 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`.

Exemplo com tipo primitivo:

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

using namespace std;

int main()
{
    vector<float> V;

    V.push_back(-4);
    V.push_back(4);
    V.push_back(-9);
    V.push_back(-12);
    V.push_back(40);

    cout << "IMPRIMINDO..." << endl;
    for (int i=0; i<V.size(); i++)
        cout << V[i] << endl;

    sort (V.begin(), V.end());
    cout << "IMPRIMINDO EM ORDEM..." << endl;
    for (int i=0; i<V.size(); i++)
        cout << V[i] << endl;

    return 0;
}
```

Exemplo utilizando classe `Pessoa` (ver exemplo anterior) criada pelo programador:

```
.....

bool ordena_por_nome(const Pessoa &A, const Pessoa &B)
{
    return (A.getNome() < B.getNome());
}

int main()
```

```

{
    vector <Pessoa> VP;
    vector <Pessoa>::iterator ptr;

    VP.push_back(Pessoa("Joao", 25));
    VP.push_back(Pessoa("Maria", 32));
    VP.push_back(Pessoa("Carla", 4));
    VP.push_back(Pessoa("Abel", 30));

    sort ( VP.begin(), VP.end(), ordena_por_nome );

    // percorrendo a lista com um ITERATOR
    for(ptr = VP.begin(); ptr != VP.end(); ptr++)
    {
        cout << "Nome: " << ptr->getNome();
        cout << " - Idade: " << ptr->getIdade() << endl;
    }
}

```

Exemplo de algoritmo com dado não STL (array C primitivo):

```

#include <iostream>
#include <algorithm>

using namespace std;

int main()
{
    double A[6] = { 1.2, 1.3, 1.4, 1.5, 1.6, 1.7 };
    reverse(A, A+6); // Passando ponteiros como iterators
    for (int i = 0; i < 6; ++i)
        cout << "A[" << i << "] = " << A[i] << endl;
    return 0;
}

```

List (lista)

- List é um tipo de contêiner sequencial que trabalha com operações de inserção e exclusão de elementos em qualquer posição do contêiner.
- É implementada como uma lista duplamente encadeada.
- Suporta iteradores de acesso bidirecional, o que permite percorrer uma lista para frente ou para trás.
- O operador subscrito [] não é suportado pela list, que só permite acesso sequencial.
- Os métodos frequentemente utilizados são:
 - empty() – testa se o contêiner está vazio.
 - clear() – limpa o contêiner (apaga todos os elementos).
 - size() – retorna o tamanho (número de elementos do contêiner).
 - begin() – retorna iterator para o primeiro elemento do contêiner.
 - end() – retorna iterator para depois do último elemento do contêiner.
 - 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.

- o `insert(posição, elemento)` – insere novo elemento antes da posição (iterator).
- o `erase(posição)` – exclui elemento na posição (iterator) especificada.
- o `erase(posição inicial, posição final+1)` – exclui elementos da faixa especificada.
- o `remove(valor)` – remove elementos com o valor dado (utiliza o operador `==`).
- o `unique()` – remove elementos duplicados consecutivos (utiliza o operador `==`).
- o `unique(funcão)` – remove elementos consecutivos para os quais função(A,B) retorna `true`.
- o `sort()` – para listas, o método `sort` deve ser utilizado ao invés do algoritmo genérico `sort` do STL. Ordena toda a lista usando o operador `<`.
- o `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).

Exemplo:

```
#include <iostream>
#include <list>
#include <algorithm>

using namespace std;

int main()
{
    list<double> minhaLista;    // cria uma lista de floats vazia
    list<double>::iterator k;   // cria um iterador de float

    minhaLista.push_back(7.5);
    minhaLista.push_back(27.26);
    minhaLista.push_front(-44);    // inserindo no início da lista
    minhaLista.push_front(7.5);    // inserindo no início da lista
    minhaLista.push_back(69.09);

    // vai imprimir seis elementos {7.5, -44, 7.5, 27.26, 69.09}
    cout << "Imprimindo a lista:\n";
    for ( k = minhaLista.begin(); k != minhaLista.end(); k++ )
        cout << *k << endl;

    // insere -2.888 como ultimo elemento
    minhaLista.insert( minhaLista.end(), -2.888);

    // retira o elemento -44 da lista
    minhaLista.remove(-44);

    // ordena a lista, em ordem ascendente
    minhaLista.sort();

    // remove elementos duplicados da lista (no caso, 7.5 aparece 2x)
    minhaLista.unique();

    // deve imprimir quatro elementos {-2.888, 7.5, 27.26, 69.09}
    cout << "Lista final ordenada....:\n";
    for ( k = minhaLista.begin(); k != minhaLista.end(); k++ )
        cout << *k << endl;

    // para usar find, informe o inicio e final+1, mais o elemento
    // este método STL devolve um iterador para o objeto.
    k = find(minhaLista.begin(), minhaLista.end(), 27.26);
    if( k == minhaLista.end() )
        cout << "Não existe o elemento procurado!!!" << endl;
    else
```

```

    cout << "Elemento 27.26 encontrado!!!" << endl;

    if (minhaLista.empty())
        cout << "Lista vazia!" << endl;
    else
        cout << minhaLista.size() << " elementos na lista!" << endl;

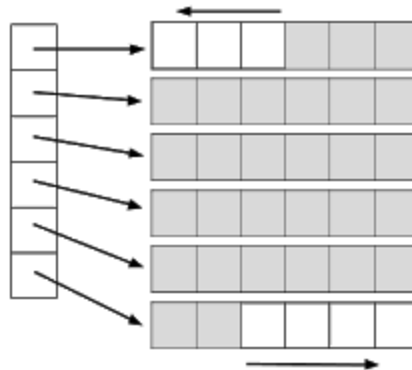
    minhaLista.clear(); // limpa toda a lista

    return 0;
}

```

Deque

- Double-Ended QUEue = fila com duas extremidades.
- Permite acesso direto aos elementos através do operador subscripto [].
- 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 do que vetores, mas podem crescer mais eficientemente sob 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 deque apresentam um desempenho pior do que as listas porém melhor do que os vetores, já que eles podem escolher o melhor sentido (para frente ou para trás) para reposicionar os elementos que precisarem ser movidos.
- 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_back()` – exclui último 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 subscripto [].
- 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 subscripto [] 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 subscripto [] 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 subscripto [] 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 itens tipo `string`.
- Cada elemento de um map é um par (`pair`). O campo `first` acessa a chave e o campo `second` acessa o item.
- Os algoritmos de busca (`find`) fazem a procura pela chave.

Exemplo:

```
#include <map>
#include <string>
#include <iostream>

using namespace std;

int main()
{
    map<string, int> digitos;
    map<string, int>::iterator iter; // o nosso iterator

    digitos["zero"] = 0;
    digitos["um"] = 1;
    digitos["dois"] = 2;
    digitos["três"] = 3;
    digitos["quatro"] = 4;
    digitos["cinco"] = 5;
    digitos["seis"] = 6;
    digitos["sete"] = 7;
    digitos["oito"] = 8;
    digitos["nove"] = 9;

    for (iter = digitos.begin(); iter != digitos.end(); iter++) {
        cout << iter->first << " = " << iter->second << endl;
    }
    // Deve imprimir (em ordem alfabética das chaves):
    // cinco = 5
    // dois = 2
    // nove = 9
    // ...
    // três = 3
    // um = 1
    // zero = 0

    return 0;
}
```

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

OPERAÇÃO	vector	deque	list	stack	queue	priority queue	set
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