

Estruturas de Informação

Biblioteca STL

Departamento de Engenharia Informática (DEI/ISEP)

Fátima Rodrigues

mfc@isep.ipp.pt

Programação Genérica

"Generic programming is a subdiscipline of computer science that deals with finding abstract representations of efficient algorithms, data structures, and other software concepts, and their systematic organization..."

Generic programming focuses on representing families of concepts"

A contribuição mais relevante e mais representativa de programação genérica é a **Biblioteca STL, C++**

STL - *Standard Template Library*

- É uma biblioteca disponibilizada para ambiente de desenvolvimento em C++ contendo um conjunto de classes de *coleções* (estruturas de dados) e de *algoritmos de coleções*, organizados na forma de *templates*
- O uso da STL oferece os seguintes benefícios:
 - **Reutilização de código** – como é baseado em templates, as classes STL podem ser adaptadas a tipos distintos sem mudança de funcionalidade (*type safe collections*)
 - **Portabilidade e facilidade de uso**

Categorias de classes na STL

Três principais componentes da Biblioteca STL:

- **Classes *containers* ≈ Contentores**

Usados para armazenar coleções de objetos

- ***Iterators* ≈ Iteradores**

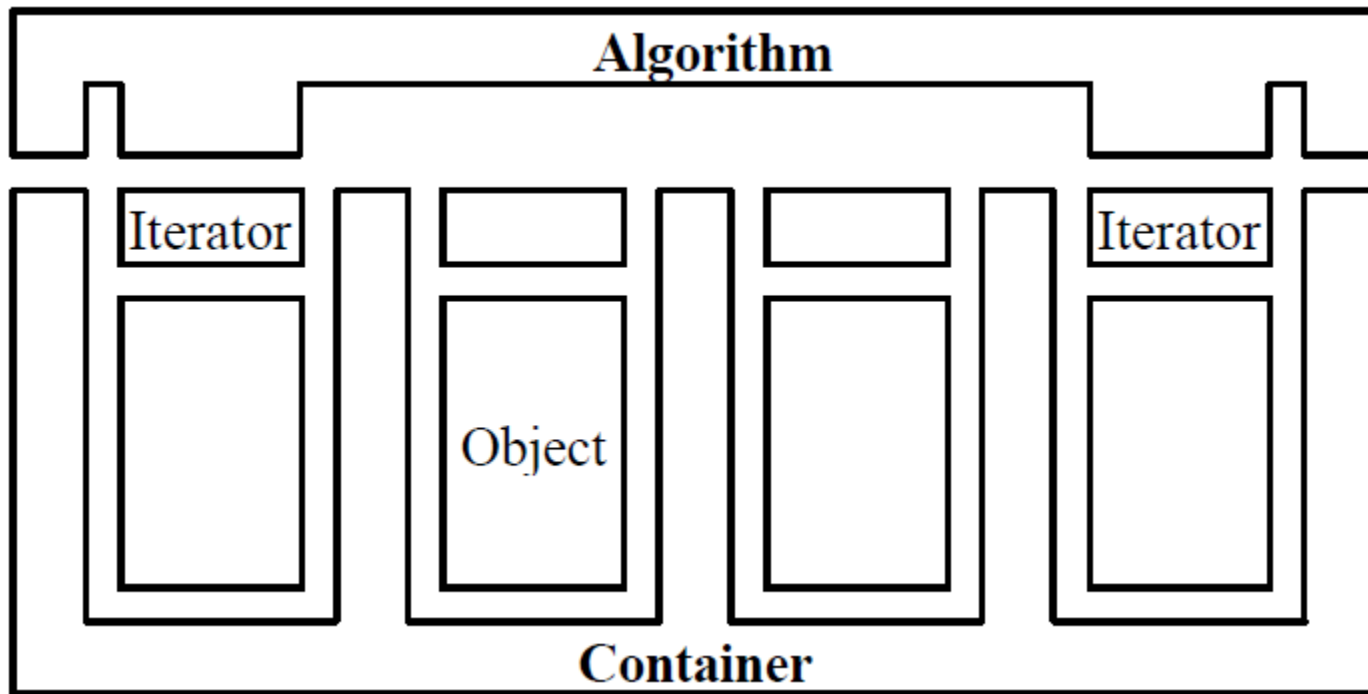
Permitem percorrer os elementos das coleções, independentemente do seu tipo, **estabelecem a ligação entre os algoritmos genéricos e os contentores**, cada contentor tem o seu próprio iterador que sabe como percorrê-lo

- **Algoritmos genéricos**

Oferecem funcionalidades genéricas para manipular os dados dos contentores

Conceito fundamental da STL

Separar dados das operações



Filosofia subjacente à STL

O mesmo algoritmo pode ser aplicado a diferentes contentores de diferentes tipos de dados:

Algoritmo Sort → vector int,

Algoritmo Sort → list string, ...

int, double, char, string, ..., Classes ...

tipos de dados

algoritmos

contentores

sort, merge, search,...

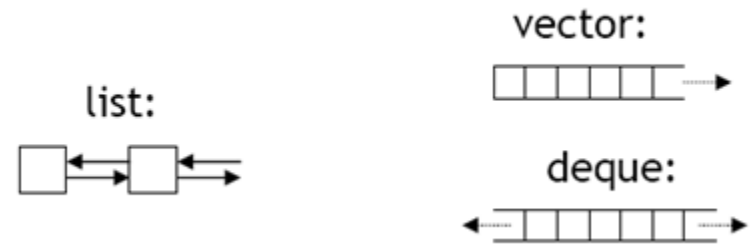
vector, deque, list,...

Contentores

Contentores correspondem a coleções de elementos de um determinado tipo

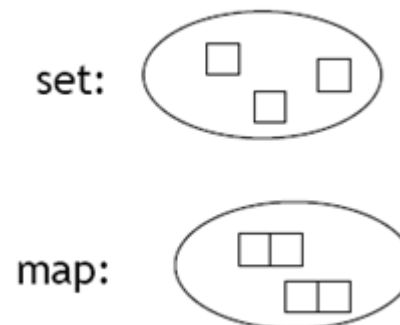
- **Contentores sequenciais**

organizam um conjunto finito de objetos, todos do mesmo tipo, de forma estritamente linear, permitindo assim o acesso sequencial aos objetos



- **Contentores associativos**

Para pares de objetos (chave, dado)



Contentores definidos na STL

Sequenciais

- **vector**: elementos organizados na forma de um *array* que pode crescer dinamicamente
- **list**: elementos organizados na forma de uma lista duplamente encadeada
- **deque**: elementos organizados em sequência, permitindo inserção ou remoção no início ou no fim sem necessidade de movimentação de outros elementos

Associativos

- **set**: coleção ordenada na qual os próprios elementos são utilizados como chaves para ordenação da coleção
- **map**: cada elemento é um par *<chave, elemento>* sendo que a *chave* é usada para ordenação da coleção
- **multiset**
- **multimap**

Contentores - Comportamento Básico

Cada contentor faz a necessária alocação de memória e possui um conjunto mínimo de operações realizadas por algoritmos genéricos da STL

- **Funções-membro comuns a todos os contentores**
insert, erase, size, empty, ...
- **Funções-membro comuns a todos os contentores sequenciais**
push_back, pop_back, ...
- **Funções-membro comuns a todos os contentores associativos**
union, intersection, difference, ...

Iteradores

Os iteradores são uma **generalização de apontadores** que permitem iterar de forma uniforme sobre os objetos contidos nos contentores

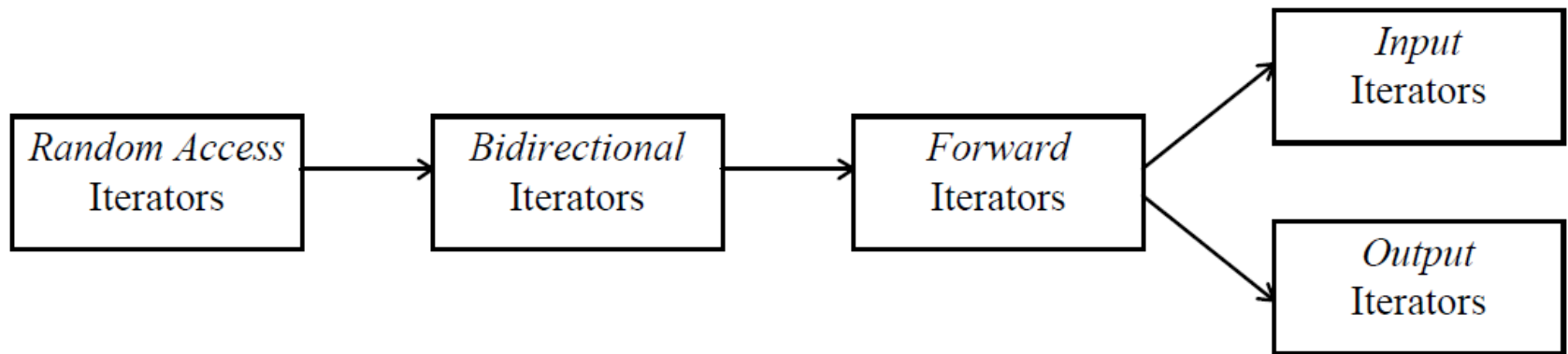
Exemplo: Percorrer de igual forma todos os elementos de um contentor, quer este seja um vector, list, ...

→ Conceito de encapsulamento

- **Iterador** é uma **espécie de ponteiro seguro** que permite que os algoritmos *Template* da biblioteca STL funcionem aplicados a diferentes contentores
- Um **iterador** representa uma certa posição dentro do contentor
- Domínio de iteradores num contentor [first, last)
first, *last* são válidos, se *last* é acessível a partir de *first*, o que significa que existe uma sequência finita de aplicações do operador ++ para *first* que faz *first* == *last*

Categorias de Iteradores

- Existem cinco categorias de iteradores de acordo com as operações que estes disponibilizam



- As categorias da esquerda satisfazem todos os requisitos das categorias à direita

Operações Iteradores

Output Iterator

Input/Forward Iterators

Bidirectional Iterator

Random Access Iterator

- construtor
- operador atribuição
- operador desreferenciação
- operador incremento pré/pós
- operador igualdade/desigualdade
- operador decremento pré/pós
- operador + (int)
- operador += (int)
- operador - (int)
- operador -= (int)
- operador -
- operador [] (int)
- operador < , > , >= , <=

Algoritmos

- Todos os algoritmos fornecidos pela biblioteca STL são **parametrizados** por tipos **iterador** e portanto são independentes das particularidades dos contentores - **algoritmos genéricos**
- Quase todos os algoritmos recebem uma gama (range) definida por dois iteradores
- Alguns algoritmos aceitam funções definidas pelo utilizador, que serão aplicadas aos elementos tratados

Tipos de Algoritmos

- **Algoritmos estáveis**

Não alteram a ordem dos elementos no contentor

for_each, find, adjacent_find, count, mismatch, equal, search, ...

- **Algoritmos de modificação**

copy, swap(vários), transform, replace(vários), fill, remove(vários),

....

- **Algoritmos de ordenação**

sort(vários), nth_element, merge, minimum/maximum,

next_permutation, ...

- **Algoritmos numéricos**

accumulate, inner_product, partial_sum, adjacent_difference,

STL - Algoritmos

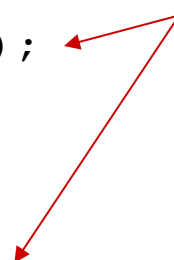
```
#include<iostream>
#include<vector>
#include<list>
#include<algorithm>
using namespace std ;

template <class InputIterator, class T>
T accumulate (InputIterator first, InputIterator last, T init)
{   while (first != last)
        init = init + *first++;
    return init ;   }

void main ()
{
    vector<int> v(5) ;
    v.push_back(3) ; ....
    int s=0 ;
    accumulate(v.begin() ,v.end() ,s) ;

    list<double> L ;
    L.push_back(5.5) ; ....
    double sum=0 ;
    accumulate(L.begin() ,L.end() ,sum) ;
}
```

**O mesmo algoritmo usado
em diferentes contentores:
Vector e Lista, com
diferentes tipos de dados**



Adaptadores

A ideia dos adaptadores é disponibilizar funcionalidades específicas de uma estrutura de dados, sem no entanto a implementar.

A estrutura é fornecida por um dos contentores definidos na STL e o seu **comportamento é *adaptado***.

Existem três adaptadores na STL:

- **stack:** fornece funcionalidade de pilha (estrutura LIFO, *last in first out*)
- **queue:** fornece funcionalidade de fila FIFO (*first in first out*).
- **priority_queue:** fornece a funcionalidade de uma fila FIFO porém com prioridades de inserção

Class vector

vector

- Um array é uma coleção de elementos todos do mesmo tipo diretamente acedidos através de um índice inteiro
- A **classe vector** é uma alternativa à estrutura de dados standard array
- Um **vector** é um **array redimensionável**
- Os vetores são mais poderosos do que os arrays devido ao número de funções disponíveis na classe

vector

- Suporta uma sequência finita de elementos
- Permite acesso aleatório rápido aos seus elementos
- Inserção e remoção de elementos no final em $O(1)$
- Inserção e remoção de elementos no início e no meio em $O(n)$
- Pesquisa em $O(n)$
- Gestão automática de memória
- Iteradores ficam inválidos após realocação de memória, inserção e remoção no meio

Idioma Array versus Idioma STL

//Idioma Array

```
void main()
{   int vecSize = 8;
    vector<int> fib(vecSize);

    fib[0] = 0;
    fib[1] = 1;

    for (int i=2; i < fib.size(); i++)
        fib[i] = fib[i-2] + fib[i-1];

    for (int i=0; i < fib.size(); i++)
        cout << fib[i] << " ";
    cout << "\n..." << endl ;
}
```

```
> 0 1 1 2 3 5 8 13
> ...
```

//Idioma STL

```
void main ()
{   vector<int> fib ;
    vector<int>::iterator it ;
    int val;

    fib.push_back(0) ;
    fib.push_back(1) ;

    for (int i=2; i < 8; i++)
    {   val=fib.back() ; val += fib[i-2] ;
        fib.push_back(val) ; }

    for (it=fib.begin(); it!=fib.end(); it++)
        cout << *it << " ";
    cout << "\n..." << endl ;
}
```

Métodos da Classe vector

Construtores

```
vector<double> v           // construtor
vector<double> x(5,10.5)   // construtor com parâmetros
vector<double> z(v)        // construtor cópia
```

Modificadores:

```
v.push_back(1.5)           //insere no fim do vetor
v.pop_back()               //retira o último elemento
v.insert(itr,12.5)         //insere onde itr estiver posicionado, os itens à dirta de
                           itr são deslocados à dirta, itr fica posiconado no item
                           inserido
v.swap(x)                  //troca o conteúdo do vetor v pelo vetor x, os vetores têm
                           de ser do mesmo tipo, podem ter tamanhos diferentes
v.clear()                  //elimina todos os eltos do vetor, o contentor fica com size=0

v.erase(v.begin()+5)       //retira o 6º elto
v.erase(v.begin(),v.begin()+3) //retira os três primeiros eltos
```

Métodos da Classe vector

Acesso aos elementos

`v.at(i)` //Devolve uma referência para o elemento na pos. i. Se i
`v[i]` não está dentro dos limites do vetor lança uma exceção
out_of_range

`v.front()`=12.5 \approx `v[0]`=12.5 //Primeiro el^{to}
`v.back()` //Último el^{to}

Capacidade

`v.size()` //Devolve o nº de elementos no vetor
`v.capacity()` //Dimensão do espaço de armazenamento atualmente atribuído ao
vetor, expressa em termos de elementos.
`v.empty()` //Verifica se o vetor está vazio, ou seja, se size = 0

Métodos da Classe vector

Iteradores:

<code>begin</code>	<code>//coloca o iterador a apontar para o início do vector</code>
<code>end</code>	<code>//coloca o iterador a apontar para o fim do vector</code>
<code>rbegin</code>	<code>//retorna um iterador inverso apontar para o último <code>el^{to}</code> do vetor</code>
<code>rend</code>	<code>//retorna um iterador inverso apontar para o 1º <code>el^{to}</code> do vetor</code>

Operadores:

<code>x = v</code>	<code>//copia todos os elementos de v para x. Se v maior que x é feita realocação de memória</code>
<code>x == v</code>	<code>//comparam o size dos vetores, se forem iguais os <code>el^{tos}</code> são</code>
<code>x != v</code>	<code>comparados sequencialmente, e pára na primeira incompatibilidade.</code>
<code>x < v</code>	
<code>x <= v</code>	
<code>x > v</code>	
<code>x >= v</code>	

Class deque

(double ended queue)

Deque

- Double-ended é um contentor sequencial que pode ser expandido ou contraído em ambas as extremidades (no início ou no fim)
- Ambos os contentores vector e deque fornecem uma interface muito semelhante e podem ser utilizados para fins semelhantes, mas internamente funcionam de forma bastante diferente
- Ao contrário dos vetores, a deque não armazena todos os seus elementos em posições de memória contíguas, mas fornece acesso direto a qualquer um dos seus elementos em tempo constante e com uma interface sequencial uniforme

Deque

- *deque* é como um vetor, mas com inserção eficiente na frente
- É um contentor que suporta acesso aleatório aos seus elementos
- Inserção e remoção de elementos no meio em $O(n)$
- Pesquisa em $O(n)$
- Inserção e remoção de elementos na frente e no final em $O(1)$
- Gestão automática de memória
- Iteradores ficam inválidos após realocação de memória, inserção e remoção no meio

Métodos da classe deque

Construtores

```
deque<double> d  
deque<double> d(4,100)  
deque<double> q(d)
```

Modificadores:

```
assign  
push_back  
push_front  
pop_back  
pop_front  
insert  
erase  
swap  
clear  
emplace  
emplace_front  
emplace_back
```

Acesso aos elementos

```
operator[]  
at  
front  
back
```

Capacidade

```
size  
max_size  
resize  
empty  
shrink_to_fit
```

Métodos da classe deque

Iteradores:

```
begin  
end  
rbegin  
rend  
cbegin  
cend  
crbegin  
crend
```

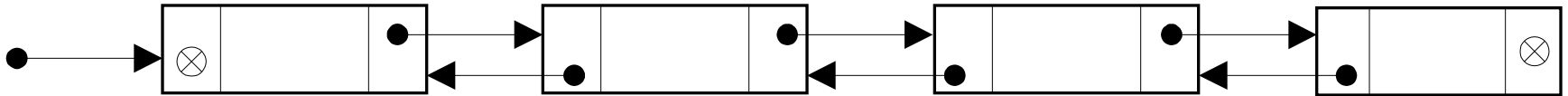
Operadores:

```
bool operator ==  
bool operator !=  
bool operator <  
bool operator <=  
bool operator >  
bool operator >=
```

Class list

(Lista Ligada)

list



- list é uma lista duplamente ligada e logo suporta listagem dos elementos para frente e para trás
- Pesquisa em $O(n)$
- Uma vez a posição encontrada, assegura inserção e eliminação (tempo constante).
- Iteradores válidos após inserção ou remoção

Métodos da classe list

Construtores

```
list<double> l  
list<double> p(5,10)  
list<double> q(p)
```

Operações:

```
splice  
remove  
remove_if  
unique  
merge  
sort  
reverse
```

Modificadores:

```
assign  
emplace_front  
push_front  
pop_front  
emplace_back  
push_back  
pop_back  
emplace  
insert  
erase  
swap  
resize  
clear
```

Métodos da classe list

Acesso aos elementos

front

back

Capacidade

empty

size

max_size

Iteradores

begin

end

rbegin

rend

cbegin

cend

crbegin

crend

Acesso aos Elementos

front

back

STL - Contentores & Iteradores

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

void main()
{
    list<char> L;
    char pt ;

    cout << "Ponto " ; cin >> pt ;
    while (pt != '*')
    {
        L.push_back(pt) ;
        cout << "Ponto "; cin >> pt; }

    cout << "Acresc. no inicio da lista " ; cin >> pt ;
    L.push_front(pt) ;

    cout << "\nLista construida: " << endl;
    for (list<char>::iterator it=L.begin(); it != L.end(); it++)
        cout << *it << " " ;
}
```

no instanciamento do iterador
deve ser fornecido o contentor
sobre o qual irá actuar

Class set

(Conjunto)

Set

- set é uma coleção ordenada de objetos do tipo “key” sem duplicados
- Operações de conjunto como interseção, união e diferença são eficientes
- Implementado por uma árvore balanceada (Red Black, Splay)
- Pesquisa em $O(\log n)$
- multiset é a versão que permite duplicados

Métodos da classe set

Construtores

```
set<int> first;  
set<int> second (100,150);  
set<int> third (second);
```

Modificadores

```
insert  
erase  
swap  
clear  
emplace  
emplace_hint
```

Acesso aos elementos

```
operator[]  
at
```

Operações

```
find  
count  
lower_bound  
upper_bound  
equal_range
```

Capacidade

```
empty  
size  
max_size
```

Iteradores

```
begin  
end  
rbegin  
rend  
cbegin  
cend  
crbegin  
crend
```

Class map

map

- *map* associa objetos do tipo “key” a objetos do tipo “data”
- Nenhum **par** de elementos possui a mesma chave
- Estrutura ordenada
- Indicada para implementação de dicionários
- Implementado por uma árvore balanceada de busca (*Red Black*, *Splay*)
- Busca em $O(\log n)$
- *multimap* é a versão que permite duplicados

Métodos da Classe map

Modificadores

insert
erase
swap
clear
emplace
emplace_hint

Operações:

find
count
lower_bound
upper_bound
equal_range

Capacidade

empty
size
max_size

Acesso aos Elementos

operator[]
at

Acesso aos elementos

front
back

Iteradores

begin
end
rbegin
rend
cbegin
cend
crbegin
crend

STL - Contentores & Iteradores

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

using namespace std ;

void main ()
{
    string word, sig ;
    map<string, string> dicion ;

    cout << " Palavra -> " ; cin >> word ;
    while (word != ".")
    {
        cout << " Significado -> " ; cin >> sig ;
        dicion[word] = sig ;
        cout << " Palavra -> " ; cin >> word ;
    }

    cout << " Dicionário " << endl ;
    map<string, string>::iterator it ;

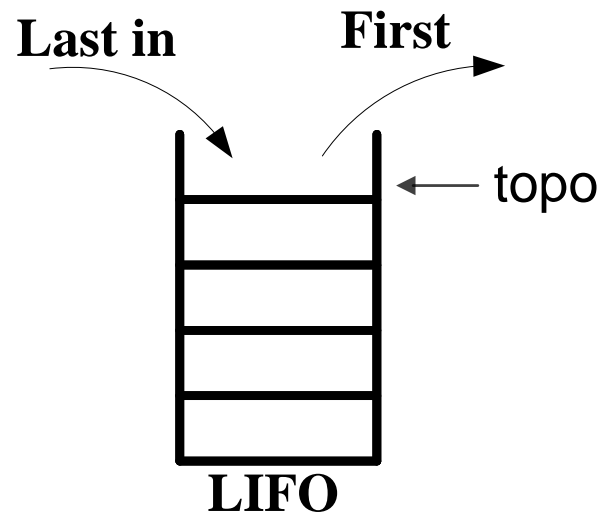
    for (it = dicion.begin(); it != dicion.end(); it++)
        cout << it->first << ' ' << it->second << endl ;
}
```


Adaptadores

Class stack

stack

- os elementos são colocados e retirados por um único lado - referenciado por **top**
- sempre que um elemento é adicionado ou retirado da pilha o topo é alterado
- estrutura de dados com uma ordenação "**LIFO - Last In First Out**"



As pilhas são usadas:

- na avaliação de expressões numéricas
- na recursividade
- nos compiladores (passagem de parâmetros, valores das variáveis locais, valores de retorno)
-

stack

- Um adaptador stack utiliza, por padrão, um container **deque** para sua implementação

Métodos:

- **push:** empilha
- **pop:** desempilha
- **empty:** verifica se está vazia
- **top:** obtém valor do topo da pilha

stack

```
#include <iostream>
#include <stack>

int main () {

    stack<int> mystack;
    int sum=0;

    for (int i=1; i<=10; i++)
        s.push(i);

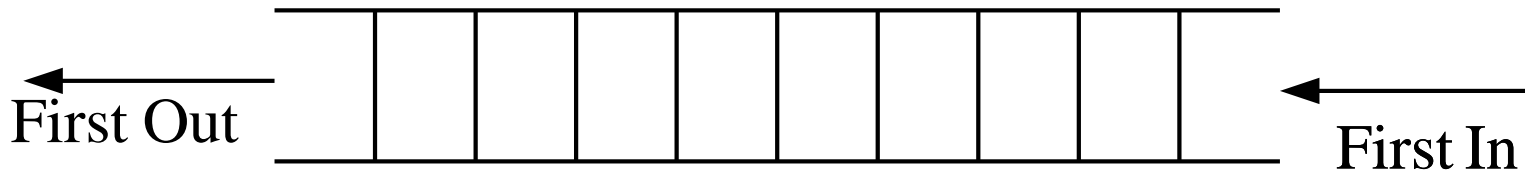
    while (!s.empty()) {
        sum += s.top();
        s.pop(); }

    cout << "total: " << sum << '\n';
    return 0;
}
```

Class queue

queue

- o acesso aos elementos é feito usando ambos os lados da fila
- no início da lista são retirados elementos
- no fim da lista são adicionados elementos
- os elementos permanecem na fila na mesma ordem em que chegaram, obtém-se uma ordenação do tipo "**FIFO - First In, First Out**"



FIFO

As filas são usadas na implementação de listas de espera:

- estudos de simulação
- escalonamento de tarefas num sistema operativo

queue

- Um adaptador queue utiliza, por padrão, um container **deque**. Pode também ser implementado com **list**.

Métodos:

- empty
- size
- front
- back
- push_back
- pop_front

queue

```
#include <iostream>
#include <queue>

int main () {
    int elementos[] = {3, 8, 4, 7, 2, 1};
    queue <int> fila;

    for (int i = 0; i < 6; i++)
        fila.push(elementos[i]);

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

Class priority queue

priority_queue

- Fila de prioridade é uma estrutura de dados que mantém uma coleção de elementos, cada um com uma prioridade associada
- O adaptador `priority_queue` utiliza, por padrão, um container **vector** para sua implementação. Pode também ser implementado com **deque**.
- A inserção é feita de acordo com a prioridade dos elementos inseridos.

Métodos:

- `empty()`
- `size()`
- `front()`
- `push_back()`
- `pop_back()`

priority_queue

```
#include <iostream>
#include <queue>

int main () {

    priority_queue<int> mypq;

    pq.push(30);
    pq.push(100);
    pq.push(25);
    pq.push(40);

    cout << "Popping out elements...";
    while (!pq.empty()) {
        cout << ' ' << pq.top();
        pq.pop();
    }
    return 0;
}
```

Popping out elements... 100 40 30 25