



**PCS3111**

**Laboratório de Programação  
Orientada a Objetos para  
Engenharia Elétrica**

**Aula 11: Namespaces, Templates e Biblioteca Padrão**

Escola Politécnica da Universidade de São Paulo

# Agenda

1. Namespaces
2. Templates
3. *Standard Template Library* (STL)
4. Container Vector
5. Container List
6. Iteradores

# Namespaces

# Projeto

- Em um projeto real existem *diversas* classes
  - Classes do projeto
  - Bibliotecas externas e de apoio
    - *Exemplo*: biblioteca padrão, acesso a dispositivos, log, etc.
- O desenvolvimento pode ser dividido em diversos “pedaços” (componentes)
  - Diversos desenvolvedores
- **Como garantir que desenvolvedores distintos não usem os mesmos nomes de classes?**
  - *Exemplo*: Classe Janela (objeto de uma casa ou tela de um programa)

# Namespace

- Espaço de nomes

- O nome de uma classe precisa ser único no namespace

Classe Janela está  
no namespace  
casa

```
4 namespace casa {  
5  
6 class Janela {  
...  
16 };  
17  
18 }
```

EX01

← Define um namespace  
com nome casa

- Em geral são definidos em arquivos de header (.h)

- Várias classes podem estar em um namespace

```
4 namespace casa {  
5  
6 class Televisao {  
...  
16 };  
17  
18 }
```

Televisao.h

# Formas de uso

## 1. Usando o operador de resolução de escopo

```
...  
5  #include "Janela.h"  
...  
8  int main() {  
9      casa::Janela *j1 = new casa::Janela();  
10     j1->fechar();  
...
```

Nome *qualificado*

EX01

## 2. Usando a diretiva using

```
...  
5  #include "Janela.h"  
6  using namespace casa;  
7  
8  int main() {  
...  
14     Janela *j2 = new Janela();  
15     j2->abrir();  
...
```

Com o using, é só usar o nome da classe!

# Detalhes

- Caso não seja definido o namespace de uma classe, ela fica em um *namespace global*
  - Ela não terá um nome qualificado
- No C++, as classes da biblioteca padrão estão no namespace std
  - É por isso que precisa-se fazer *using namespace std* para usar o cout!

# Templates



# Problema

## ■ Considere a classe *Pilha*

```
4 class Pilha {
5 public:
6     void push(int valor);
7     int pop();
8 private:
9     int valores[10];
10    int topo = 0;
11 };
```

EX02

```
3 void Pilha::push(int valor) {
4     valores[topo] = valor;
5     topo++;
6 }
7
8 int Pilha::pop() {
9     topo--;
10    return valores[topo];
11 }
```

- E se eu quiser uma pilha de *string* ou de *Pedido*?

```
6 class PilhaDePedido {
7 public:
8     void push(Pedido* valor);
9     Pedido* pop();
10 private:
11     Pedido* valores[10];
12     int topo = 0;
13 };
```

Trocar int  
por Pedido\*

```
8 class PilhaDeString {
9 public:
10    void push(string valor);
11    string pop();
12 private:
13    string valores[10];
14    int topo = 0;
15 };
```

# Problema

- Copy & paste!
  - O código é *praticamente* o mesmo
    - O que muda é apenas o tipo
- Como resolver esse problema?
  - **Conceito de Template**

# Template

- É uma forma de polimorfismo
  - Também chamado de *classe parametrizável* ou *generics*

- Definição

 Ou *typename*

```
template <class NomeDoTipo> class SuaClasse {  
    ...  
}
```

- O *NomeDoTipo* deve ser usado na sua classe como se fosse um tipo normal

- Uso

- Necessário dizer qual é o tipo escolhido

```
SuaClasse<TipoEscolhido> a;
```

- Pode ser uma classe ou mesmo um tipo primitivo

# Exemplo

T é o tipo a ser substituído

```
4  template <class T> class Pilha {
5  public:
6      void push(T valor);
7      T pop();
8  private:
9      T valores[10];
10     int topo = 0;
11 };
12
13 template <class T>
14 void Pilha<T>::push(T valor) {
15     valores[topo] = valor;
16     topo++;
17 }
18
19 template <class T>
20 T Pilha<T>::pop() {
21     topo--;
22     return valores[topo];
23 }
```

Métodos em função de T

EX03

Pilha de string

```
9  int main() {
10     Pilha<string>* p1 =
        new Pilha<string>();
11     p1->push("a");
12     p1->push("b");
    ...
}
```

Pilha de Pedido\*

```
15     Pilha<Pedido*>* p2 =
        new Pilha<Pedido*>();
16     p2->push (new Pedido(1));
17     p2->push (new Pedido(2));
    ...
}
```

# Template em C++

- Funcionamento do template
  - Quando o compilador lê a palavra `template` no código fonte, não gera código
    - Ainda não sabe o tipo
  - Quando o objeto é instanciado, o compilador sabe o tipo do argumento e **instancia** o template
    - Ou seja, gera o código com o tipo correspondente
  - Isto é repetido para cada vez em que se instancia o objeto

# ***Standard Template Library (STL)***

# Biblioteca padrão do C++

- C++ possui uma biblioteca padrão
  - Algoritmos e tipos já implementados
    - ...“reinventar a roda”...
  - Disponível em qualquer compilador C++
    - Portabilidade
- *Exemplo*
  - Tipos (string, números complexos, etc.)
  - Funções matemáticas (sqrt, log, sin, tan, etc.)
  - Entrada e saída
  - Relógio e tempo

# ***Standard Template Library***

- Parte da biblioteca padrão
- Define **containers**
  - Estruturas de dados básicas
  - Pilha, fila, lista ligada, tabelas hash, etc.
- Define **algoritmos**
  - Algoritmos básicos em containers
    - Ordenação, busca, etc.
- Usam o conceito de *template*
- **Já estão implementadas**
  - É só usar!

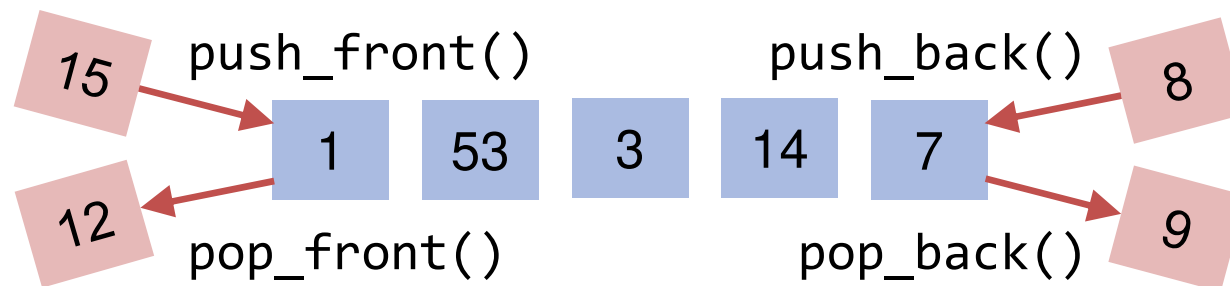


# Containers

- Estruturas **dinâmicas**
  - Crescem e diminuem conforme a necessidade
- Divididos em 2 tipos
  - **Sequências:** sequências de elementos
    - `vector`: vetor redimensionável
    - `list`: lista duplamente ligada
    - `forward-list`: lista ligada simples
    - `deque`: permite inserir e remover tanto no começo quanto no fim
  - **Containers associativos:** permitem obter um valor a partir de uma chave (são *tabelas hash* e *árvores*)
    - Tipos: `map`, `multimap` e `set`

# Alguns métodos básicos

Método (<T> é o tipo parametrizado)	Descrição	vector	list	deque	forward -list
int size()	Obtêm o número de elementos	X	X	X	
bool empty()	Informa se o container está vazio	X	X	X	X
<T> back()	Obtêm o <i>último</i> elemento	X	X	X	
void push_back(<T>)	Adiciona um elemento no <i>fim</i>	X	X	X	
void pop_back()	Remove o <i>último</i> elemento	X	X	X	
<T> front()	Retorna o <i>primeiro</i> elemento	X	X	X	X
void push_front(<T>)	Adiciona um elemento no <i>início</i>		X	X	X
void pop_front()	Remove o <i>primeiro</i> elemento		X	X	X



# Algoritmos

- *Alguns* algoritmos disponíveis
  - Necessário fazer `#include <algorithm>`

Algoritmo	Descrição
<code>find</code>	Encontra o primeiro elemento <i>equivalente</i> ao valor
<code>count</code>	Conta o número de elementos que possuem o valor
<code>equal</code>	Compara o conteúdo de dois <i>containers</i>
<code>search</code>	Procura por uma sequência de valores que corresponde à mesma sequência em outro <i>container</i>
<code>swap</code>	Trocar um valor em um local por outro
<code>sort</code>	Ordena o <i>container</i>
<code>merge</code>	Combina os elementos de dois <i>containers</i>

# Vector

# Vector

- Vetor redimensionável
  - Controla o número de elementos
    - Aumenta e diminui de tamanho automaticamente
  - Está no namespace `std`
    - Necessário `#include <vector>`

Ao usar classes, prefira usar ponteiros (senão a atribuição pode ficar *confusa*)

- Possui **apenas** métodos para inserir e remover no **fim** (questão de eficiência)
  - `push_back()` e `pop_back()`

# Métodos de acesso

## ■ Operador [ ]

- Acessa uma posição de um vector como se fosse um vetor

```
8   vector<Pedido*>* v = new vector<Pedido*>();  
9   v->push_back(new Pedido(1));  
10  cout << (*v)[0]->getId() << endl;
```

EX04

## • Cuidados

- Trabalha com **valores** (e não **ponteiros**)!
- Só acesse uma posição depois de “inicializa-la” com um `push_back`

## ■ <T> at(int posicao)

- Obtêm o valor em posicao do vector

# Exemplo

```
16  vector<Pedido*>* pedidos = new vector<Pedido*>();
17  pedidos->push_back(new Pedido(100));
18  pedidos->push_back(new Pedido(101));
19
20  for(unsigned int i = 0; i < pedidos->size(); i++) {
21      cout << pedidos->at(i)->getId() << endl;
22  }
23
24  // Apagando
25  while (!pedidos->empty()) {
26      Pedido *ultimo = pedidos->back();
27      pedidos->pop_back();
28      delete ultimo;
29  }
30  delete pedidos;
```

Saída

100  
101

Se não for unsigned,  
gera um *warning*

EX04

Veja em <http://www.cplusplus.com/reference/vector/vector/>  
os outros métodos de vector

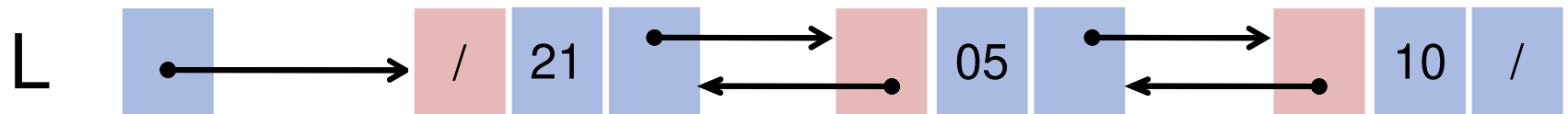
# List



# List

- É uma lista **duplamente ligada**

- Está no namespace `std`
- Necessário `#include <list>`



- Não é necessário saber os detalhes de implementação da lista
  - É só usar os métodos!
- Permite inserir no começo e no fim
  - `push_front`, `push_back`, `pop_front` e `pop_back`

# Exemplo

EX05

Saída

4 3 1 2

```
2  #include <list>
3  #include "Pedido.h"
4
5  using namespace std;
6
7  int main() {
8      list<Pedido*>* pedidos = new list<Pedido*>();
9      pedidos->push_back(new Pedido(1));
10     pedidos->push_back(new Pedido(2));
11
12     pedidos->push_front(new Pedido(3));
13     pedidos->push_front(new Pedido(4));
14
15     while (!pedidos->empty()) {
16         Pedido *p = pedidos->front();
17         cout << p->getId() << " ";
18         pedidos->pop_front();
19         delete p;
20     }
21
22     return 0;
23 }
```

Inserindo no fim

Inserindo no início

# Containers

- Os métodos `at` e o operador `[]` não estão disponíveis!
  - **Então como varrer um list?**
    - *Problema:* usar os ponteiros internos...
- É importante definir uma **interface comum** para varrer **containers diferentes!**
  - Senão os algoritmos de apoio serão específicos para cada container!
- **Solução:** iterador

# Iteradores

# Iterador

- Forma **homogênea** para varrer um container
  - **Independente** se é vector, list, map, forward\_list etc.
- É um **tipo**
  - Usado por todas as estruturas de dados da STL
- Métodos **comuns** em containers para **usar** iteradores
  - `iterator begin()`
    - Obtêm o iterador para o início da estrutura
  - `iterator end()`
    - Obtêm o iterador com o fim da estrutura
    - Usado para saber se acabou a varredura

# Iterador

- Funciona como se fosse um **ponteiro**
  - Operador ++
    - Usado para ir para a próxima posição
  - Operador \*
    - Obtêm o valor do iterador
- **Cuidado:** o iterador é para um tipo específico
  - Iterador de *vetor de inteiros*

# Exemplo

## ■ Usando um iterator com list

```
7  list<Pedido*>* pedidos = new list<Pedido*>();
8  pedidos->push_back (new Pedido(1));
9  pedidos->push_back (new Pedido(2));
10
11  pedidos->push_front(new Pedido(3));
12  pedidos->push_front(new Pedido(4));
13
14  list<Pedido*>::iterator i = pedidos->begin();
15
16  while (i != pedidos->end()) {
17      cout << (*i)->getId() << " ";
18      i++;
19  }
```

EX06

Saída

4 3 1 2

Obtendo o iterator  
para o início

Enquanto não se  
chegar no fim

Obtendo o valor

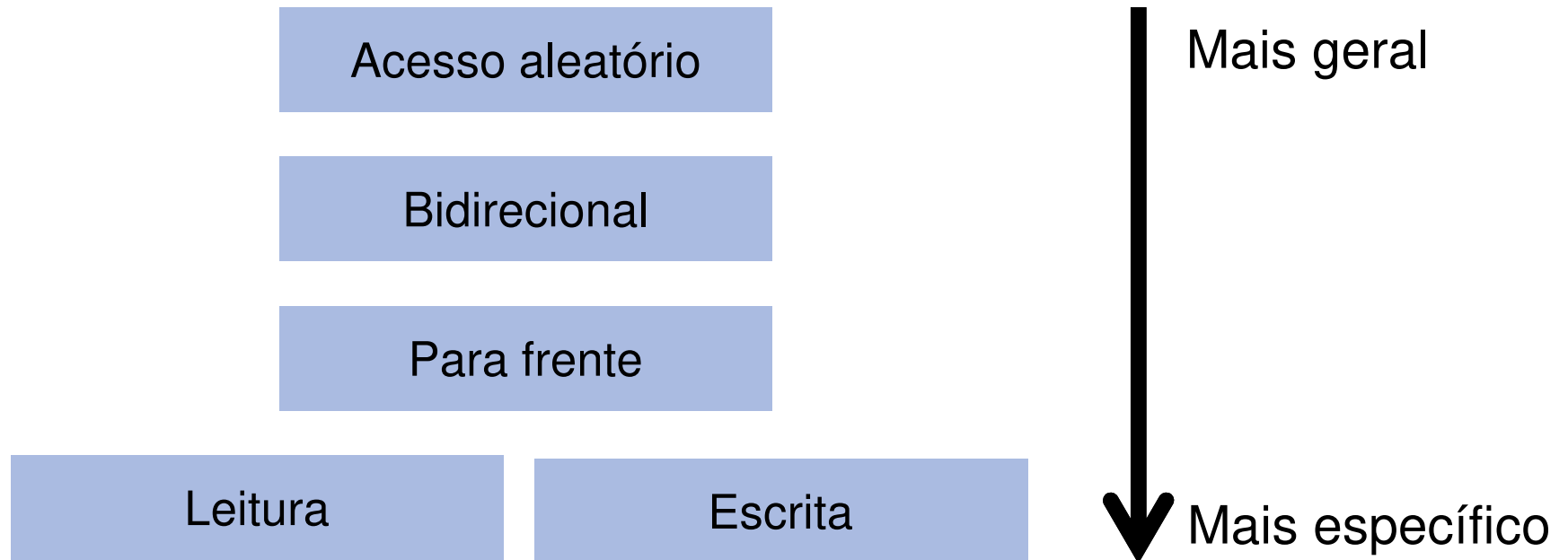
Passando para o  
próximo elemento

# Tipos de iterador

- Todos os iteradores permitem ir para o próximo elemento (`i++`)
- Mas nem todos permitem **escrita**, ir para o **anterior** e acesso **aleatório**!
  - Escrita: `*i = valor`
  - Anterior: `i--`
  - Acesso aleatório: `i[1]`



# Tipos de iterador



Tipo	Vector	List	Deque	Set	Multiset	Map	Multimap
Acesso aleatório	X		X				
Bidirecional	X	X	X	X	X	X	X
Para frente	X	X	X	X	X	X	X
Escrita	X	X	X	X	X	X	X
Leitura	X	X	X	X	X	X	X

# Tipos de iterador

- Algoritmos precisam de tipos específicos

Algoritmo	Leitura	Escrita	Para frente	Bidirecional	Acesso Aleatório
find	X				
count	X				
sort					X
merge	X	X			
reverse				X	
replace			X		
unique			X		

- Consulte o que o algoritmo precisa antes de usá-lo!
  - <http://www.cplusplus.com/reference/algorithm/>

# Exemplo

EX07

```
1  #include <iostream>
2  #include <list>
3  #include <algorithm>
4
5  using namespace std;
6
7  int main() {
8      list<int>* valores = new list<int>();
9      ...
14     list<int>::iterator valor =
15         find(valores->begin(), valores->end(), 8);
16
17     if (valor != valores->end() )
18         cout << "Encontrou o " << *valor << endl;
19     else cout << "Nao encontrou" << endl;
```

Chamando o find

O find usa o operador == para encontrar. Para procurar um objeto a partir de um atributo específico, será necessário *sobrecarregar* esse operador!

Mais detalhes em <https://isocpp.org/wiki/faq/operator-overloading>

# Bibliografia

- LAFORE, R. **Object-Oriented Programming in C++**. 4th ed., SAMS, 2002.
  - Capítulo 14