# Estruturas de Dados I

## Vistas, Iteradores e Percorríveis

Igor Machado Coelho

03/11/2026

1 Vistas, Iteradores e Percorríveis

2 Agradecimentos

Section 1

# Vistas, Iteradores e Percorríveis

# Pré-Requisitos

São requisitos para essa aula o conhecimento de:

- Introdução/Fundamentos de Programação (em alguma linguagem de programação)
- Interesse em aprender C/C++
- Noções de tipos de dados e ponteiros
- Noções de conceitos (`concept`) em C++, vistos nas aulas de Pilhas e Filas

# Motivação

Até agora, nossos TADs (Pilha, Fila, Conjunto) **guardam** seus elementos: cada um é **dono** dos seus dados.

Mas como **olhar** para os elementos de uma estrutura, um por um, sem copiá-los e sem conhecer como ela foi implementada?

Nesta aula, definiremos três componentes básicos, que usaremos em todas as próximas estruturas:

- **Vista:** uma referência leve a um valor guardado em outro lugar
- **Iterador:** uma vista que **pode se mover** pelos elementos
- **Percorrível:** algo que fornece um iterador para seus elementos

Eles existem, com outros nomes, em todas as linguagens modernas!

# Conceito: Vista

Uma *Vista* (do inglês *View*) é um TAD que **referencia** um valor guardado em outro lugar (a *origem*), com três propriedades: **não-posse:** alterações na origem são percebidas pela vista; **leve:** custo constante $O(1)$; **validade:** dura enquanto a origem existir.

```
template <typename V>
concept VistaTAD = requires(V v) {
   typename V::Tipo;    // tipo associado
   { v.le() };          // leitura origem
};

struct VistaInt {
   using Tipo = int;    // tipo associado
   const Tipo* origem; // não muda origem
   Tipo le() { return *origem; }
};
static_assert(VistaTAD<VistaInt>);
```

```
// Exemplo de uso
int x = 10;
VistaInt y{.origem = &x};
println("{}", y.le()); // 10

x = 30;
println("{}", y.le()); // 30 (não-posse)
```

**Desafio:** por que `origem` é do tipo `const int*` e não `int*`?

# Conceito: Vista Mutável

Uma *Vista Mutável* é uma Vista que também possui `altera(v)`, que modifica o valor **na origem**.

```
template <typename V>
concept VistaMutavelTAD = VistaTAD<V> &&
   requires(V v, typename V::Tipo t) {
   { v.altera(t) };  // escrita na origem
};

struct VistaMutavelInt {
   using Tipo = int;  // tipo associado
   Tipo* origem;
   Tipo  le()          { return *origem; }
   void altera(Tipo v) { *origem = v; }
};
static_assert(VistaMutavelTAD<VistaMutavelInt>);
```

```
// Exemplo de uso
int x = 10;
VistaMutavelInt y{.origem = &x};

y.altera(20);
println("{}", x);       // 20

x = 30;
println("{}", y.le()); // 30 (não-posse)
```

**Desafio:** `VistaMutavelInt` satisfaz `VistaTAD`? E `VistaInt` satisfaz `VistaMutavelTAD`?

# Conceito: Vista de Vetor

Uma *Vista de Vetor* é uma vista de **vários** elementos consecutivos, lidos pela posição `i`. A *Vista Mutável de Vetor* também altera o elemento, **na origem**.

```
template <typename V, typename T>
concept VistaVetorTAD = requires(V v, int i) {
   { v.tamanho() } -> std::same_as<int>;
   { v.le(i) }     -> std::same_as<T>;
};
template <typename V, typename T>
concept VistaMutavelVetorTAD =
   VistaVetorTAD<V, T> &&
   requires(V v, int i, T t) {
   { v.altera(i, t) };
};  // le e altera: pre(0 <= i < tamanho())

// o SpanInt da introdução já serve!
static_assert(VistaMutavelVetorTAD<SpanInt, int>);
```

```
// dobra os elementos, na origem
void dobra(
   VistaMutavelVetorTAD<int> auto v) {
   for (int i = 0; i < v.tamanho(); i++)
      v.altera(i, 2 * v.le(i));
}
// Exemplo de uso
int a[] = {1, 2, 3};
SpanInt s{.inicio = a, .n = 3};
dobra(s);                  // cópia leve
println("{} {} {}",
   a[0], a[1], a[2]);  // 2 4 6
```

# Conceito: Iterador

Um *Iterador* (do inglês *Iterator*) é uma **Vista** do elemento atual que **pode se mover**: além de `le()`, possui `proximo()` e `terminou()`.

```
template <typename It, typename T>
concept IteradorTAD = VistaTAD<It> &&
   std::same_as<typename It::Tipo, T> &&
   requires(It it) {
   { it.terminou() };  // fim da origem?
   { it.proximo() };   // avança
};
struct IteradorVetorInt {
   using Tipo = int;
   const Tipo* atual;  // posição atual
   const Tipo* fim;    // após o último
   Tipo le()       { return *atual; }
   bool terminou() { return atual == fim; }
   void proximo()  { atual++; }
};
```

```
// Exemplo de uso
int v[3] = {10, 20, 30};
IteradorVetorInt it{.atual = v,
                    .fim = v + 3};
while (!it.terminou()) {
   println("{}", it.le()); // 10 20 30
   it.proximo();
}
```

# Conceito: Percorrível

Um *Percorrível* (do inglês *Iterable* ou *Range*) é um TAD com uma única operação: `iterador()`, que retorna um **Iterador** posicionado no primeiro elemento e permite visitar os demais.

```
template <typename P, typename T>
concept PercorrivelTAD = requires(P p) {
   { p.iterador() } -> IteradorTAD<T>;
};
struct TrioInt {
   using Tipo = int;
   Tipo dados[3];
   IteradorVetorInt iterador() {
      return {.atual = dados,
              .fim = dados + 3};
   }
};
static_assert(PercorrivelTAD<TrioInt, int>);
```

```
// Exemplo de busca no trio
auto buscaTrio(auto it, int x) {
   while(!it.terminou() && it.le() != x)
      it.proximo();
   return it; // terminou() se não achou
}
// Exemplo de uso
TrioInt t{.dados = {10, 20, 30}};
println("{}",
    buscaTrio(t.iterador(), 20).le());
println("{}",
    buscaTrio(t.iterador(), 99).terminou
```

**Desafio:** qual o custo de busca? Seria possível fazer melhor?

# Vistas, Iteradores e Percorríveis na Computação

Os mesmos conceitos aparecem, com outros nomes, nas linguagens modernas:

| Curso | C++ | Java | Python | Posse? |
|---|---|---|---|---|
| Vista | `std::string_view`, `std::span` | - | `memoryview` | não |
| Iterador | iteradores (`*`, `++`, fim) | `Iterator` | iterador (`next`) | não |
| Percorrível | *range* (`begin`/`end`) | `Iterable` | *iterable* (`iter`) | pode ter |

Note que um Percorrível **pode** ser dono dos elementos (um `std::vector` é um *range*), mas Vistas e Iteradores **nunca** são: são leves, e por isso podem ser passados **por valor**.

***Observação:*** como nossos TADs não são copiáveis, algoritmos genéricos (como `buscaTrio`) recebem o **iterador**, e não o Percorrível!

# Exercício (para casa) / reflexão

Implemente um Iterador Mutável.

```
template <typename It, typename T>
concept IteradorMutavelTAD = IteradorTAD<It, T> && VistaMutavelTAD<It>;
```

Quais métodos ele teria? Para que serviria?

Outra reflexão: Vistas são TAD muito simples e não carregam recursos dinâmicos (*non-owning*), portanto **não terão** métodos `cria()` e `libera()` nesse curso.

# Tópico Avançado: Vista Verificável

A validade da Vista pode ser verificada pelo compilador (Clang 23+) com **construtor** cujo parâmetro seja anotado com `lifetimebound`, ligando a vida da Vista à vida da origem.

```
#if __has_cpp_attribute(clang::lifetimebound)
#define LFTM [[clang::lifetimebound]]
#else
#define LFTM  // outros compiladores: ignora
#endif

struct VistaVerificavelInt {
   using Tipo = int;
   const Tipo* origem;
   VistaVerificavelInt(const Tipo* x LFTM)
      : origem(x) {}
   Tipo le() { return *origem; }
};
static_assert(VistaTAD<VistaVerificavelInt>);
```

```
// Exemplo de uso (com erro!)
int z = 0;
VistaVerificavelInt y{&z};
{
   int x = 10;
   y = VistaVerificavelInt(&x);
}  // x morre aqui...
println("{}", y.le()); // ... y ainda usa
```

```
$ clang++ -std=c++20 -Wlifetime-safety ...
warning: local variable ‘x’ does not
live long enough
note: local variable ‘x’ is destroyed here
note: later used here
```

**Desafio:** e com a `VistaInt` (sem construtor), o compilador avisa? Por quê?

# Vistas, Iteradores e Percorríveis: Revisão Geral

- O que significa dizer que uma Vista **não tem posse**? Quando ela deixa de ser válida?
- Qual a diferença entre uma Vista e uma Vista Mutável?
- Por que um Iterador **é** uma Vista? O que ele acrescenta?
- O que um Percorrível precisa oferecer? Ele pode ser dono dos seus elementos?
- Por que algoritmos genéricos recebem o iterador, e não o Percorrível?

**Próxima aula:** o **Saco**, nosso primeiro TAD Percorrível, implementado com listas encadeadas e vetores!

# Bibliografia Recomendada

Além da bibliografia do curso, recomendamos para esse tópico:

- Szwarcfiter, J.L; Markenzon, L. Estruturas de Dados e seus Algoritmos. Rio de Janeiro, LTC, 1994.

Bibliografia Adicional:

- Sedgewick, R.; Wayne, K. Algorithms, 4th Edition. Addison-Wesley, 2011.
- Livro "The C++ Programming Language" de Bjarne Stroustrup
- Dicas e normas C++: https://github.com/isocpp/CppCoreGuidelines

Section 2

# Agradecimentos

# Pessoas

Em especial, agradeço aos colegas que elaboraram bons materiais, como o prof. Fabiano Oliveira (IME-UERJ), e o prof. Jayme Szwarcfiter cujos conceitos formam o cerne desses slides.

Estendo os agradecimentos aos demais colegas que colaboraram com a elaboração do material do curso de Pesquisa Operacional, que abriu caminho para verificação prática dessa tecnologia de slides.

# Software

Esse material de curso só é possível graças aos inúmeros projetos de código-aberto que são necessários a ele, incluindo:

- pandoc
- LaTeX
- GNU/Linux
- git
- markdown-preview-enhanced (github)
- visual studio code
- atom
- revealjs
- groomit-mpx (screen drawing tool)
- xournal (screen drawing tool)
- ...

# Empresas

Agradecimento especial a empresas que suportam projetos livres envolvidos nesse curso:

- github
- gitlab
- microsoft
- google
- ...

# Reprodução do material

Esses slides foram escritos utilizando pandoc, segundo o tutorial ilectures:

- https://igormcoelho.github.io/ilectures-pandoc/

Exceto expressamente mencionado (com as devidas ressalvas ao material cedido por colegas), a licença será Creative Commons.

**Licença:** CC-BY 4.0 2020

Igor Machado Coelho

# This Slide Is Intentionally Blank (for goomit-mpx)