

M A T E R I A L D E R E V I S Ã O

Implementação UML em C++

Conceitos essenciais para a prova de TecProg

const

static

List & Vector

Iterators

Destruição dinâmica

namespace

O que vamos revisar

01	const	Atributos, parâmetros e métodos const	<i>p. 3-5</i>
02	static	Atributos e métodos de classe — inicialização e uso	<i>p. 6-8</i>
03	List vs Vector	Quando usar cada container da STL	<i>p. 9-11</i>
04	Iterators	.begin(), .end() e como percorrer containers	<i>p. 12-14</i>
05	Destruição dinâmica	new / delete e destrutoras corretas	<i>p. 15-17</i>
06	namespace	Organização e using namespace	<i>p. 18-19</i>
+	Tópicos extras	Polimorfismo, operadores, static_cast, herança	<i>p. 20-25</i>

O que significa const?

const é uma promessa ao compilador: "isto não vai mudar". Ele bloqueia em tempo de compilação qualquer tentativa de modificação.

1. Parâmetro const

```
void f(const int x)
```

Garante que a função não altera o valor recebido. Visto em `Mito(const bool v = true)` e `setVivo(const bool v)`.

2. Retorno const

```
const int getId() const
```

O valor retornado não pode ser alterado pelo chamador. Comum em todos os getters das provas.

3. Método const

```
int getId() const { ... }
```

O const APÓS a assinatura promete que o método não modifica atributos do objeto (`this`).

Método const: o detalhe que cai na prova

Por que TODOS os getters são const?

Porque você precisa conseguir chamar o getter mesmo a partir de uma referência ou ponteiro const. Se o método não for const, o compilador recusa a chamada.

Exemplo real (Prova T13 — Inseto::operator>):

```
class Inseto {  
    public:  
        const int getEnergia() const { return energia; }  
        // método const pode chamar OUTRO método const  
        const bool operator>(Inseto& I) const {  
            return (getEnergia() > I.getEnergia());  
        }  
};
```

REGRA DE OURO Se um método NÃO modifica o objeto, ele DEVE ser const. Em prova de UML, esquecer o const após a assinatura é erro frequente.

Atributos const: só na lista de inicialização

Quando o atributo é const, ele NÃO pode ser atribuído no corpo da construtora — apenas inicializado na lista de inicialização (depois dos dois pontos).

ERRADO**Atribuição no corpo da construtora**

```
class SE {  
    private:  
        const short int num_manual;  
        const short int num_eletronica;  
    public:  
        SE() {  
            num_manual = 5;        // erro!  
            num_eletronica = 10;    // erro!  
        }  
};
```

CORRETO**Inicialização na lista (Prova T16)**

```
class SE {  
    private:  
        const short int num_manual;  
        const short int num_eletronica;  
    public:  
        SE(Partido* p = NULL):  
            num_manual(5),  
            num_eletronica(10),  
            urnas(),  
            partidos(p)  
        { ... }  
};
```

Atributo e método estáticos

Um membro static pertence à CLASSE, não a cada objeto. Existe uma única cópia compartilhada por todas as instâncias.

Visualização



Útil para: contadores compartilhados, registros globais (lista de KingKongs vivos), ponteiros para classes que coordenam o sistema (SE).

Atributo static: declarar dentro, definir FORA

Regra fundamental: dentro da classe você só DECLARA o atributo static. A DEFINIÇÃO (que aloca a memória) tem de ficar fora da classe, no escopo global.

Exemplo 1 — Contador (Prova T14, classe Mito):

```
class Mito {  
    private:  
        static int cont;           // declaração  
    public:  
        Mito(const bool v = true) : id(cont++), vivo(v) { } // pós-incremento!  
};  
int Mito::cont(0);                // definição FORA da classe
```

Exemplo 2 — Ponteiro static (Prova T16, classe Urna):

```
class Urna {  
    protected:  
        static SE* pS;           // declaração  
};  
SE* Urna::pS = NULL;             // definição (sempre FORA)
```

Incrementar corretamente e método static

Incrementar ao criar cada objeto

Use `cont++` (pós-incremento) na lista de inicialização.
O objeto recebe o valor ATUAL e o contador é incrementado em seguida.

```
Mito() : id(cont++), vivo(true) { }  
  
// 1º objeto: id = 0,  cont = 1  
// 2º objeto: id = 1,  cont = 2  
// 3º objeto: id = 2,  cont = 3
```

Cuidado: `++cont` (pré-incremento) faria o primeiro id ser 1, não 0. Verifique o que o enunciado pede.

Método static

Pertence à classe — chama-se SEM ter um objeto.
Não pode acessar atributos não-static (não há `this`).

```
// Prova T14 – KingKong  
static void considerar(KingKong* pk) {  
    if (pk) kks.push_back(pk);  
}  
  
// Chamada: SEM instanciar objeto  
KingKong::considerar(novoKK);  
  
// Prova T16 – Urna  
Urna::setSE(&sistema);
```

vector, list e set: o que muda?

São containers da STL para guardar coleções de objetos. A escolha entre eles depende de COMO você acessa e modifica os elementos.

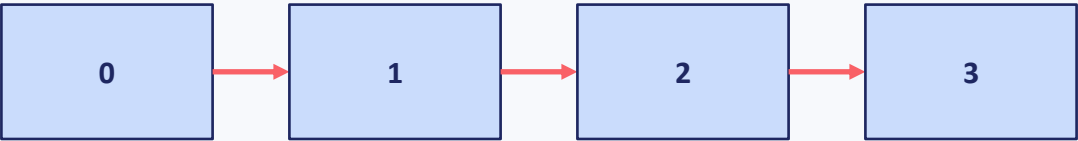
Container	Como organiza	Acesso por índice [i]	Inserção/remoção meio	Header
vector	Array dinâmico contíguo	✓ rápido $O(1)$	✗ caro $O(n)$	<vector>
list	Lista duplamente encadeada	✗ não tem	✓ rápido $O(1)$	<list>
set	Árvore ordenada, sem duplicatas	✗ não tem	✓ $O(\log n)$	<set>

Visualização da memória

vector — contíguo



list — encadeada (com ponteiros)



vector → use quando precisa de acesso por índice (urnas[i] na Prova T16).
list / set → use quando precisa percorrer com iteradores (Prova T13 usa set; T14 usa vector).

Operações que aparecem nas provas

Operação	O que faz	Exemplo das provas
<code>push_back(elem)</code>	adiciona no fim	<code>kks.push_back(pk);</code>
<code>size()</code>	número de elementos	<code>int tam = (int)kks.size();</code>
<code>empty()</code>	true se está vazio	<code>lider(kks.empty())</code>
<code>operator[i]</code>	acessa o elemento i	<code>urnas[i]->votar();</code>
<code>erase(it)</code>	remove elemento na posição it	<code>kks.erase(it);</code>
<code>clear()</code>	esvazia o vector	<code>urnas.clear();</code>
<code>begin() / end()</code>	iteradores de início/fim	<code>it = kks.begin();</code>

DICA Sempre faça `(int)kks.size()` em laços com índice — `size()` retorna `size_t` (sem sinal) e isso evita warnings na comparação com `int`.

Quando NÃO há acesso por índice

list e set não permitem container[i]. Você é OBRIGADO a usar iteradores para percorrer. Foi exatamente isso que caiu na Prova T13 com set<Inseto*>.

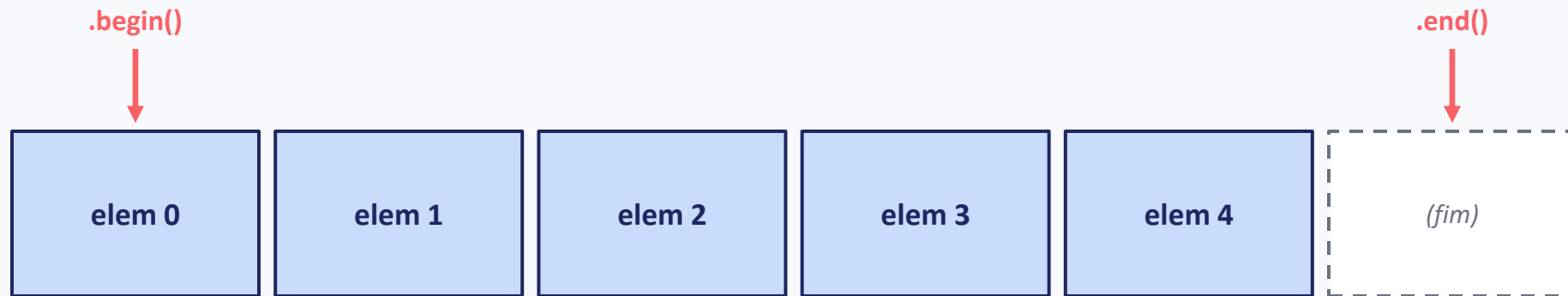
Operação	list	set	Observação
adicionar	push_back(x)	insert(x)	set não duplica, list duplica
tamanho	size()	size()	ambos têm
remover	erase(it)	erase(it)	via iterador
esvaziar	clear()	clear()	remove tudo
acesso [i]	X	X	use iterador

Trecho real da Prova T13 — Colônia<Inseto*> com set:

```
set<Inseto*> conj;    // declaração
conj.insert(pi);      // adicionar
conj.clear();          // esvaziar
```

O que são iteradores?

Um iterador é um "ponteiro inteligente" que aponta para um elemento dentro de um container. Você o move com `++` e desreferencia com `*`.



ATENÇÃO!

`.end()` NÃO aponta para o último elemento. Aponta para a posição IMEDIATAMENTE APÓS o último — uma sentinela. Por isso o teste em laços é sempre `it != container.end()` (e não `<=`).

Como percorrer um vector / list / set

Padrão 1 — for clássico (Prova T13, Colonia::executar)

```
set<Inseto*>::const_iterator iterador;  
for (iterador = conj.begin();  
     iterador != conj.end();  
     iterador++) {  
    if (*iterador) {  
        (*iterador)->executar();    // desreferencia 2x: it->ponteiro->objeto  
    }  
}
```

Padrão 2 — while com erase (Prova T14, KingKong::lutar)

```
vector<KingKong*>::iterator it = kks.begin();  
while (it != kks.end()) {  
    if (false == (*it)->getVivo()) {  
        kks.erase(it);    // erase invalida o iterador → NÃO incremente  
    } else {  
        it++;    // incrementa só se NÃO removeu  
    }  
}
```

Desreferenciar e principais armadilhas

Quando o container guarda PONTEIROS, você desreferencia DUAS vezes:

Expressão	Significado	Tipo resultante
<code>it</code>	o iterador em si	<code>iterator</code>
<code>*it</code>	o que está GUARDADO no container	<code>Inseto*</code>
<code>(*it)->metodo()</code>	chama método NO objeto apontado	depende do método
<code>it-></code>	atalho — só funciona se <code>*it</code> NÃO for ponteiro	raro nas provas

Armadilhas comuns

Usar `[i]` em set ou list

Não compila — esses containers não têm `operator[]`.

Incrementar após `erase`

`erase(it)` invalida `it`. Use a versão `while`: incrementa só no `else`.

Comparar com `<`

Iteradores de `list/set` não suportam `<`. Sempre use `!=`.

Esquecer o `*`

`for(it ...) { it->fazer(); }` só funciona se `*it` não for ponteiro.

Cada new precisa do seu delete

Em C++ não há garbage collector. Tudo que você criou com new TEM de ser destruído com delete — caso contrário, vazamento de memória.

CRIAR (Construtora)

```
// Prova T14 - Ilha::criarKings
void criarKings(const short int x) {
    for (int i = 0; i < MAX; i++) {
        if ((rand()%100) < x) {
            pkings[i] = new KingKong();
        }
    }
}
```

DESTRUIR (Destrutora)

```
// Prova T14 - Ilha::~~Ilha
~Ilha() {
    for (int i = 0; i < MAX; i++) {
        if (pkings[i]) {
            delete pkings[i];
            pkings[i] = NULL;
        }
    }
}
```

O padrão dos 3 passos para destruir

1

Verificar

if (ptr) — só destrua se o ponteiro não for nulo. delete em NULL não dá erro, mas testar é boa prática e evita ambiguidades.

2

Liberar

delete ptr; — chama a destrutora do objeto e devolve a memória ao sistema. Para vetores: delete[] ptr;

3

Anular

ptr = NULL; — evita ponteiro pendurado (dangling pointer). Sem isso, o ponteiro continua "apontando" para memória inválida.

Destruir todos os objetos de uma coleção

Quando o container tem PONTEIROS para objetos criados com new, você precisa fazer delete em cada um ANTES de limpar o container.

Com vector (índice — Prova T16)

```
SE::~SE() {
    int tam = (int)urnas.size();
    for (int i = 0; i < tam; i++) {
        if (urnas[i]) {
            delete urnas[i];
            urnas[i] = NULL;
        }
    }
    urnas.clear();
}
```

Com set (iterador — Prova T13)

```
Colonia::~Colonia() {
    set<Inseto*>::const_iterator it;
    it = conj.begin();
    while (it != conj.end()) {
        if (*it) {
            delete *it;    // libera o objeto
        }
        it++;
    }
    conj.clear();          // só DEPOIS limpa
}
```

Organizando o código em pacotes

Um namespace agrupa classes, funções e variáveis em um "pacote" lógico. Em UML, todo pacote (caixa externa do diagrama) vira um namespace em C++.

Por que usar?

 **Evitar colisão de nomes** Duas classes Inseto em projetos diferentes não brigam se estiverem em namespaces distintos.

 **Refletir o pacote UML** Provas T13 e T14 deixam explícito: "O pacote precisa se tornar namespace".

 **Organização** Agrupa elementos relacionados — Mundo, Mitologico, etc.

DETALHE DA PROVA

A Prova T16 traz a marca "Pacote que NÃO se torna namespace" — leia sempre o enunciado. Nem todo pacote vira namespace. T13 e T14 viram; T16 não.

Sintaxe: declarar e usar

Declarar (Prova T13 — Mundo)

```
namespace Mundo {  
    class Inseto { ... };  
    class Abelha { ... };  
    class Formiga { ... };  
    class Colonia { ... };  
    class Habitat { ... };  
} using namespace Mundo;
```

Usar dentro do namespace e em main()

```
// Forma 1: qualificado  
Mundo::Habitat habitat;  
// Forma 2: usando o namespace  
using namespace Mundo;  
Habitat habitat;  
// main da Prova T13  
int main() {  
    Habitat habitat;  
    return 0;  
}
```

TRUQUE DAS PROVAS

Repare em `} using namespace Mitologico;` — fecha o namespace E aplica using na mesma linha. Atalho comum nos enunciados.

P A R T E 2

Outros tópicos que aparecem

nas resoluções e que valem ser comentados antes da prova

- Classe abstrata + virtual puro

- Polimorfismo (virtual + ponteiro base)

- Sobrecarga de operadores

- `static_cast` (upcast seguro)

- Visibilidade: `public` / `protected` / `private`

- Lista de inicialização da construtora

Classe abstrata e método virtual puro

Uma classe é abstrata quando tem ao menos um método virtual puro (= 0). NÃO PODE ser instanciada diretamente — só serve como base para herdeiras.

Em UML

Estereótipo <<Abstrata>>

Nome da classe em
itálico

Métodos virtuais puros também em itálico, com
= 0

Sintaxe (Prova T13)

```
class Inseto {  
    private:  
        virtual void variar() = 0;  
    public:  
        virtual ~Inseto() {  
            energia = -1;  
        }  
        virtual void executar() = 0;  
};
```

Regras

- Não é possível **Inseto** x;
- Mas pode ter ponteiro:
Inseto*
- Toda derivada DEVE implementar o método virtual puro, senão também é abstrata
- Destrutora
virtual é obrigatória nesses casos

virtual + ponteiro de classe-base

Polimorfismo: chamar um método pelo PONTEIRO da classe-base e o C++ escolhe AUTOMATICAMENTE qual implementação rodar (a da derivada).

3 ingredientes obrigatórios

1 virtual na base

```
virtual void executar() = 0;
```

2 Ponteiro / referência da base

```
Inseto* pi;
```

3 Override na derivada (mesma assinatura)

```
void executar() { variar(); }
```

Em ação (Prova T13 — Colonia::executar)

```
for (iterador = conj.begin(); iterador != conj.end(); iterador++) {  
    (*iterador)->executar();    // Inseto* aponta para Abelha OU Formiga  
}  
// → se for Abelha, roda Abelha::executar()  
// → se for Formiga, roda Formiga::executar()  
// quem decide é o virtual + o objeto real, em tempo de execução
```

Sobrecarga: operator>, ++, --

Você pode definir o que operadores como > , ++ , -- significam para uma classe sua. As provas pedem isso direto no diagrama UML.

Operador	Assinatura típica	Vem das provas
<code>operator></code>	<code>const bool operator>(Classe& c) const</code>	T13 (Inseto), T14 (Alado), T16 (Partido)
<code>operator++</code>	<code>void operator++()</code>	T16 — Partido (incrementa votos)
<code>operator--</code>	<code>void operator--()</code>	T13 — Inseto (decrementa energia)

Três formas equivalentes de chamar (Prova T13 — `Formiga::interagir`)

```
// 1) sintaxe natural
if (getEnergia() > pAbelha->getEnergia())
// 2) chamada explícita do operator
if (operator>(*pAbelha))
// 3) usando *this
if ((*this) > (*pAbelha))
```

static_cast: o cast seguro do C++

`static_cast<Tipo>(expr)` é o cast checado em tempo de compilação. Use sempre que precisar tratar um ponteiro de derivada como ponteiro de base (upcast).

Upcast: Abelha* → Inseto* (Prova T13)

```
Abelha* pAbelha = new Abelha();  
pColonia->incluir(static_cast<Inseto*>(pAbelha));  
// incluir(Inseto* pi) espera ponteiro da BASE  
// static_cast diz ao compilador: "trate como Inseto*"
```

Mesma ideia na Prova T14 — Ilha::combater

```
g.lutar(static_cast<Mito*>(pkings[i])); // KingKong* → Mito*  
a.lutar(static_cast<Mito*>(pkings[i])); // idem
```

public, protected, private em UML e C++

Símbolo UML	C++	Quem acessa?	Onde aparece nas provas
+	public	qualquer um	construtoras, getters/setters, executar()
#	protected	a própria classe e suas DERIVADAS	T14: vivo (Mito) e informar(); T16: id, percentual, pS (Urna)
—	private	SÓ a própria classe	atributos id, energia, ferrao, pAbelha — quase tudo

Quando usar protected?

Quando o atributo/método precisa ser usado pela classe base E suas derivadas, mas NÃO pode ficar acessível a estranhos. Exemplo: `informar()` em Mito é chamado por `Godzilla::lutar` e `KingKong::lutar`.

DETALHE PRECIOSO

Um objeto da classe X PODE acessar atributos privados de OUTRO objeto da MESMA classe X — basta passá-lo como parâmetro. É assim que `Inseto::operator>` consegue chamar `I.getEnergia()`. Cai na T16 também.

Lista de inicialização

A lista de inicialização vem após : (dois pontos) na construtora e ANTES das chaves. É onde se inicializa: const, referências, atributos sem construtora padrão e a classe-base.

Anatomia (Prova T14 — Alado)

```
Alado(const bool v = true) :  
    Godzilla(v),           // 1) chama construtora da BASE  
    idoso(getIdade() > 700) // 2) inicializa atributo próprio  
{  
    sementear();           // 3) corpo (atribuições já não-iniciais)  
}
```

Quando você É OBRIGADO a usar a lista de inicialização:

- Atributos const
- Chamar construtora da classe-base
- Atributos do tipo referência (&)
- Atributos cujo tipo não tem construtora padrão

Checklist antes de entregar a prova



Toda construtora inicializa TODOS os atributos (lista ou corpo)



Atributos const e referências estão na LISTA de inicialização



Métodos que não modificam o objeto terminam com const



Toda destrutora libera o que foi alocado com new e zera o ponteiro



Cada new tem seu delete correspondente (sem vazamento)



Atributos static estão DECLARADOS na classe e DEFINIDOS fora



Métodos virtuais puros = 0 estão implementados nas derivadas



Destrutora da base é virtual quando há herança polimórfica



Iteradores comparados com !=, nunca com <



Após erase(it), não incrementa o iterador



O pacote UML virou namespace (a menos que o enunciado diga o contrário)



static_cast<Base*>() usado nos upcasts pedidos