Tutorial de C++ para Programadores C# - Versão Aprofundada

Introdução

Este tutorial foi desenvolvido para programadores experientes em C# que desejam aprender C++, com foco em conceitos intermediários e avançados. Ao longo deste material, exploraremos as principais características de C++, comparando-as com seus equivalentes em C# sempre que possível.

O tutorial aborda tanto recursos do C++11 quanto do C++23, permitindo que você compreenda a evolução da linguagem e utilize os recursos mais modernos disponíveis.

Público-alvo

Este tutorial é destinado a programadores senior de C# .NET 9 que desejam expandir seus conhecimentos para C++. Assumimos que você já possui:

- Sólido conhecimento de programação orientada a objetos
- Experiência com C# e o ecossistema .NET
- Compreensão de conceitos como gerenciamento de memória, tipos de referência e valor
- Familiaridade com recursos avançados de C# como LINQ, async/await, e generics

Estrutura do Tutorial

O tutorial está organizado em seções que abordam diferentes aspectos da linguagem C++, sempre fazendo comparações com C# e destacando as diferenças fundamentais entre as duas linguagens. Cada seção inclui exemplos de código comentados que podem ser compilados e executados.

1. Fundamentos Avançados

1.1. Sistema de Tipos

C++ e C# possuem sistemas de tipos fundamentalmente diferentes. Enquanto C# é uma linguagem gerenciada com garbage collection, C++ oferece controle direto sobre a memória.

Value Semantics vs. Reference Semantics Em C#, tipos de valor (structs) são armazenados na pilha, enquanto tipos de referência (classes) são armazenados no heap e gerenciados pelo garbage collector. Em C++, a distinção é menos clara:

- Em C++, tanto structs quanto classes podem ser alocados na pilha ou no heap
- A semântica de valor é o padrão em C++, mesmo para tipos complexos
- A semântica de referência em C++ é explícita, usando ponteiros ou referências

Exemplo: Structs em C++ vs C C#:

```
// Em C#, structs são tipos de valor
struct Ponto
{
    public int X { get; set; }
    public int Y { get; set; }
}

// Ao copiar, cria-se uma nova instância
```

```
Ponto p1 = new Ponto { X = 10, Y = 20 };
Ponto p2 = p1; // Cópia completa
p2.X = 30;     // Não afeta p1
```

C++:

```
// Em C++, structs são semelhantes a classes, mas com membros públicos por padrão
struct Ponto {
    int x;
    int y;
};

// Comportamento semelhante ao C# para tipos na pilha
Ponto p1 = {10, 20};
Ponto p2 = p1;  // Cópia completa
p2.x = 30;      // Não afeta p1

// Mas em C++ também podemos usar referências
Ponto& p3 = p1;
p3.x = 40;      // Modifica p1 diretamente
```

1.2. Gerenciamento de Memória

Uma das diferenças mais significativas entre C++ e C# é o gerenciamento de memória. C# utiliza garbage collection, enquanto C++ requer gerenciamento manual ou uso de técnicas como RAII e smart pointers.

RAII (Resource Acquisition Is Initialization) RAII é um padrão fundamental em C++ onde a aquisição de um recurso (como memória, arquivo, mutex) é vinculada à inicialização de um objeto, e a liberação desse recurso é vinculada à destruição do objeto.

Exemplo em C++:

```
{
    std::unique_ptr<int> ptr = std::make_unique<int>(42);
    // O recurso é gerenciado automaticamente
    // Quando ptr sair de escopo, a memória será liberada
}
```

Equivalente aproximado em C#:

```
using (var resource = new DisposableResource())
{
    // O recurso será liberado automaticamente ao final do bloco using
}
```

2. Move Semantics (Semântica de Movimento)

2.1. Conceito e Importância

Move semantics é um dos recursos mais importantes introduzidos no C++11, permitindo transferir recursos de um objeto para outro sem fazer cópias desnecessárias. Isso é particularmente útil para objetos que gerenciam recursos pesados, como grandes buffers de memória, conexões de rede ou handles de arquivos.

Em C#, não há um conceito equivalente direto, pois os objetos são gerenciados por referência e o garbage collector cuida da liberação de memória.

2.2. Rvalue References

C++ introduziu um novo tipo de referência chamado rvalue reference (referência para rvalue), denotado por &&. Isso permite distinguir entre:

- **lvalues**: expressões que se referem a um objeto identificável (como uma variável)
- **rvalues**: expressões temporárias ou valores literais

```
// Exemplo de rvalue reference
void processar(std::string&& str) {
    // str é uma referência para um rvalue (objeto temporário)
    std::cout << "Processando: " << str << std::endl;
}

std::string obterString() {
    return "Temporário";
}

int main() {
    std::string s = "Permanente";

    // Erro: não pode passar lvalue para rvalue reference
    // processar(s);

    // OK: std::move converte lvalue para rvalue
    processar(std::move(s));

    // OK: resultado de função é rvalue
    processar(obterString());

    // OK: literal de string é rvalue
    processar("Literal");

    return 0;
}
```

2.3. Implementação de Move Constructors e Move Assignment Operators

Para habilitar move semantics em suas classes, você precisa implementar:

1. **Move Constructor**: `Class(Class&& other) noexcept`
2. **Move Assignment Operator**: `Class& operator=(Class&& other) noexcept`

Exemplo completo de uma classe com move semantics:

```
class RecursoPesado {
private:
    std::string nome;
    std::unique_ptr<int[]> dados;
    size_t tamanho;

public:
    // Construtor padrão
    RecursoPesado() : nome("Sem nome"), dados(nullptr), tamanho(0) {}

    // Construtor com parâmetros
    RecursoPesado(const std::string& n, size_t tam)
```

```
        : nome(n), dados(new int[tam]), tamanho(tam) {
        // Inicializar dados...
    }

    // Construtor de cópia - operação cara
    RecursoPesado(const RecursoPesado& outro)
        : nome(outro.nome), dados(new int[outro.tamanho]), tamanho(outro.tamanho) {
        // Copiar todos os dados - operação potencialmente cara
        for (size_t i = 0; i < tamanho; ++i) {
            dados[i] = outro.dados[i];
        }
    }

    // Operador de atribuição por cópia - operação cara
    RecursoPesado& operator=(const RecursoPesado& outro) {
        if (this != &outro) {
            // Realocar e copiar dados
            nome = outro.nome;
            dados.reset(new int[outro.tamanho]);
            tamanho = outro.tamanho;

            // Copiar todos os dados
            for (size_t i = 0; i < tamanho; ++i) {
                dados[i] = outro.dados[i];
            }
        }
        return *this;
    }

    // Construtor de movimento - operação barata
    RecursoPesado(RecursoPesado&& outro) noexcept
        : nome(std::move(outro.nome)), dados(std::move(outro.dados)), tamanho(outro.tamanho) {
        // Deixar o outro objeto em estado válido mas vazio
        outro.tamanho = 0;
    }

    // Operador de atribuição por movimento - operação barata
    RecursoPesado& operator=(RecursoPesado&& outro) noexcept {
        if (this != &outro) {
            // Mover recursos
            nome = std::move(outro.nome);
            dados = std::move(outro.dados);
            tamanho = outro.tamanho;

            // Deixar o outro objeto em estado válido mas vazio
            outro.tamanho = 0;
        }
        return *this;
    }
};
```

2.4. std::move e Perfect Forwarding

`std::move` é uma função que converte um lvalue em um rvalue, permitindo que o objeto seja movido em vez de copiado:

```
std::vector<int> v1 = {1, 2, 3, 4, 5};
std::vector<int> v2 = std::move(v1); // v1 agora está em estado "movido"
```

Perfect forwarding permite preservar a categoria de valor (lvalue ou rvalue) ao passar argumentos através de funções template:

```
template<typename T>
void encaminhar(T&& arg) {
    processar(std::forward<T>(arg)); // Preserva a categoria de valor
}

// Uso:
std::string s = "Teste";
encaminhar(s);              // s é lvalue, permanece lvalue
encaminhar(std::move(s));  // std::move(s) é rvalue, permanece rvalue
encaminhar("Literal");     // "Literal" é rvalue, permanece rvalue
```

2.5. Move Semantics com Containers

Os containers da STL são otimizados para move semantics:

```
std::vector<RecursoPesado> vetor;

// Inserindo com cópia
RecursoPesado recurso1("Original", 10000);
vetor.push_back(recurso1); // Cria uma cópia

// Inserindo com movimento
vetor.push_back(std::move(recurso1)); // Move o recurso, recurso1 fica vazio

// Construção in-place (sem cópia ou movimento)
vetor.emplace_back("Novo", 10000); // Constrói diretamente no vetor
```

2.6. Comparação com C

C# não tem um conceito equivalente a move semantics porque:

1. Objetos são gerenciados por referência, então "mover" um objeto significa apenas copiar a referência
2. O garbage collector cuida da liberação de memória
3. Para tipos de valor (structs), C# sempre faz cópias completas

```
// Em C#, atribuir objetos apenas copia a referência
var lista1 = new List<int> { 1, 2, 3 };
var lista2 = lista1; // Ambas as variáveis apontam para o mesmo objeto

// Para tipos de valor, sempre há cópia
struct Dados {
    public int[] Valores;
}
```

```
var d1 = new Dados { Valores = new int[1000] };
var d2 = d1; // Cópia do struct, mas ambos apontam para o mesmo array
```

3. Smart Pointers

3.1. Problema dos Raw Pointers

Em C++, ponteiros brutos (raw pointers) apresentam vários desafios:

1. Não indicam propriedade do recurso
2. Não liberam automaticamente a memória
3. Podem causar vazamentos de memória se não forem gerenciados corretamente
4. Podem levar a double-free ou uso após liberação (use-after-free)

```
// Problemas com raw pointers
void funcaoProblematica() {
    int* ptr = new int(42);
    // ... código que pode lançar exceção ...
    delete ptr; // Pode nunca ser executado se houver exceção
}

// Outro problema: quem é responsável pela liberação?
int* criarArray() {
    return new int[100]; // Quem deve chamar delete[]?
}
```

3.2. std::unique_ptr

`std::unique_ptr` implementa o conceito de propriedade exclusiva - apenas um único objeto pode possuir o recurso por vez.

```
// Uso básico de unique_ptr
std::unique_ptr<int> ptr1 = std::make_unique<int>(42);
std::cout << *ptr1 << std::endl; // Acesso ao valor: 42

// Transferência de propriedade
std::unique_ptr<int> ptr2 = std::move(ptr1); // ptr1 agora é nullptr

// Arrays
std::unique_ptr<int[]> array = std::make_unique<int[]>(10);
array[0] = 1;
array[1] = 2;

// Liberação automática ao sair do escopo
```

Comparação com C#:

```
// C# não precisa de unique_ptr, pois o GC gerencia a memória
var obj = new MinhaClasse();
// obj será coletado pelo GC quando não houver mais referências
```

3.3. std::shared_ptr

`std::shared_ptr` implementa propriedade compartilhada com contagem de referências.

```
// Criação de shared_ptr
std::shared_ptr<int> ptr1 = std::make_shared<int>(42);
```

```cpp
std::cout << "Contagem: " << ptr1.use_count() << std::endl; // 1

// Compartilhamento
std::shared_ptr<int> ptr2 = ptr1;
std::cout << "Contagem: " << ptr1.use_count() << std::endl; // 2

// Liberação
ptr1.reset(); // Reduz contagem para 1
std::cout << "Contagem: " << ptr2.use_count() << std::endl; // 1

// O recurso será liberado quando a contagem chegar a 0
```

Comparação com C#:

```cpp
// Em C#, todos os objetos de classe funcionam como shared_ptr
var obj = new MinhaClasse();
var ref2 = obj; // Ambas as variáveis apontam para o mesmo objeto
// O objeto será coletado quando não houver mais referências
```

3.4. std::weak_ptr

`std::weak_ptr` é uma referência fraca que não impede a liberação do recurso. É útil para quebrar referências circulares.

```cpp
// Problema de referência circular com shared_ptr
struct Pessoa;
struct Equipe {
    std::string nome;
    std::vector<std::shared_ptr<Pessoa>> membros;
};

struct Pessoa {
    std::string nome;
    std::shared_ptr<Equipe> equipe; // Causa referência circular
};

// Solução com weak_ptr
struct PessoaCorreta {
    std::string nome;
    std::weak_ptr<Equipe> equipe; // Não aumenta contagem de referências

    void mostrarEquipe() {
        if (auto e = equipe.lock()) { // Tenta obter shared_ptr válido
            std::cout << "Equipe: " << e->nome << std::endl;
        } else {
            std::cout << "Equipe não existe mais" << std::endl;
        }
    }
};
```

Comparação com C#:

```cpp
// C# tem WeakReference para casos similares
WeakReference<MinhaClasse> weakRef = new WeakReference<MinhaClasse>(obj);

if (weakRef.TryGetTarget(out MinhaClasse target)) {
```

```
    // Usar target
} else {
    // Objeto foi coletado pelo GC
}
```

3.5. Casos de Uso Avançados

Custom Deleters Você pode personalizar como os recursos são liberados:

```
auto deleter = [](FILE* f) {
    std::cout << "Fechando arquivo" << std::endl;
    fclose(f);
};

{
    std::shared_ptr<FILE> file(fopen("data.txt", "r"), deleter);
    // Uso do arquivo...
} // deleter é chamado automaticamente
```

enable_shared_from_this Permite que um objeto obtenha um `shared_ptr` para si mesmo:

```
class Widget : public std::enable_shared_from_this<Widget> {
public:
    std::shared_ptr<Widget> getShared() {
        return shared_from_this();
    }
};

// Uso:
auto w = std::make_shared<Widget>();
auto w2 = w->getShared(); // Obtém outro shared_ptr para o mesmo objeto
```

Alocadores Personalizados

```
// Alocador personalizado
template<typename T>
class MyAllocator {
public:
    using value_type = T;
    T* allocate(std::size_t n) {
        return static_cast<T*>(::operator new(n * sizeof(T)));
    }
    void deallocate(T* p, std::size_t n) {
        ::operator delete(p);
    }
};

// Uso com shared_ptr
std::shared_ptr<int> ptr = std::allocate_shared<int>(MyAllocator<int>(), 42);
```

3.6. Comparação Detalhada com C

Característica	C++	C#
Gerenciamento de memória	Manual com smart pointers	Automático com GC
Propriedade exclusiva	std::unique_ptr	Não tem equivalente direto
Propriedade compartilhada	std::shared_ptr	Comportamento padrão para classes
Referências fracas	std::weak_ptr	WeakReference
Referências circulares	Problema com shared_ptr, resolvido com weak_ptr	Detectadas e coletadas pelo GC
Deleters personalizados	Suportados	IDisposable para limpeza explícita
Thread safety	Contagem de referências thread-safe, objeto não	Referências thread-safe

4. Templates e Metaprogramação

4.1. Templates vs. Generics

Templates em C++ e generics em C# servem propósitos similares, mas têm implementações fundamentalmente diferentes.

Templates em C++

```cpp
// Template de função
template<typename T>
T maximo(T a, T b) {
    return (a > b) ? a : b;
}

// Template de classe
template<typename T>
class Caixa {
private:
    T valor;
public:
    Caixa(T val) : valor(val) {}
    T obterValor() const { return valor; }
};

// Uso
int m1 = maximo(10, 20);
double m2 = maximo(3.14, 2.71);
Caixa<std::string> caixa("Olá");
```

Generics em C

```csharp
// Método genérico
public T Maximo<T>(T a, T b) where T : IComparable<T>
{
    return a.CompareTo(b) > 0 ? a : b;
}

// Classe genérica
public class Caixa<T>
{
    private T valor;
    public Caixa(T val) { valor = val; }
    public T ObterValor() { return valor; }
}

// Uso
int m1 = Maximo(10, 20);
double m2 = Maximo(3.14, 2.71);
Caixa<string> caixa = new Caixa<string>("Olá");
```

Principais diferenças

1. **Tempo de instanciação**:
   - C++: Templates são expandidos em tempo de compilação
   - C#: Generics são instanciados em tempo de execução
2. **Verificação de tipos**:
   - C++: Duck typing (se compila, funciona)
   - C#: Constraints explícitas (where T : IComparable)
3. **Especialização**:
   - C++: Permite especialização de templates para tipos específicos
   - C#: Não permite especialização de generics
4. **Parâmetros não-tipo**:
   - C++: Permite parâmetros que não são tipos (como inteiros)
   - C#: Apenas parâmetros de tipo

4.2. Metaprogramação com Templates

C++ permite metaprogramação em tempo de compilação usando templates, algo que não tem equivalente direto em C#.

```cpp
// Cálculo de fatorial em tempo de compilação
template<unsigned int N>
struct Fatorial {
    static constexpr unsigned int valor = N * Fatorial<N-1>::valor;
};

// Caso base
template<>
struct Fatorial<0> {
    static constexpr unsigned int valor = 1;
};

// Uso
constexpr auto fat5 = Fatorial<5>::valor; // Calculado em tempo de compilação
```

4.3. Concepts (C++20)

Concepts em C++20 são semelhantes às constraints em C#, mas mais poderosos.

```cpp
// Definição de um concept
template<typename T>
concept Numerico = std::is_arithmetic_v<T>;

// Uso de concept em uma função template
template<Numerico T>
T dobrar(T valor) {
    return valor * 2;
}
```

Equivalente aproximado em C#:

```csharp
public T Dobrar<T>(T valor) where T : struct, INumber<T>
{
    return valor + valor;
}
```

5. Expressões Lambda

5.1. Lambdas em C++

```cpp
// Lambda básica
auto soma = [](int a, int b) { return a + b; };

// Lambda com captura
int multiplicador = 5;
auto multiplica = [multiplicador](int valor) { return valor * multiplicador; };

// Lambda com captura por referência
int contador = 0;
auto incrementa = [&contador]() { return ++contador; };
```

5.2. Lambdas em C

```csharp
// Lambda básica
Func<int, int, int> soma = (a, b) => a + b;

// Lambda com captura (implícita)
int multiplicador = 5;
Func<int, int> multiplica = valor => valor * multiplicador;

// Captura por referência (usando ref local)
int contador = 0;
Func<int> incrementa = () => ++contador;
```

5.3. Principais diferenças

1. **Captura de variáveis**:
   - C++: Explícita, usando `[...]`
   - C#: Implícita, o compilador determina o que capturar
2. **Tipo de retorno**:
   - C++: Pode ser deduzido ou explícito com `-> tipo`

    - C#: Deduzido pelo compilador ou determinado pelo tipo do delegate
3. **Mutabilidade**:
    - C++: Lambdas são imutáveis por padrão, use `mutable` para permitir modificações
    - C#: Lambdas podem modificar variáveis capturadas

6. Concorrência e Paralelismo

6.1. Modelo de Memória

C++ e C# têm modelos de memória diferentes para concorrência.

Modelo de Memória em C++

```cpp
std::atomic<bool> pronto(false);
std::atomic<int> dados(0);

// Thread produtor
std::thread produtor([&]() {
    dados.store(42, std::memory_order_release);
    pronto.store(true, std::memory_order_release);
});

// Thread consumidor
std::thread consumidor([&]() {
    while (!pronto.load(std::memory_order_acquire)) {
        std::this_thread::yield();
    }
    int valor = dados.load(std::memory_order_acquire);
});
```

Modelo de Memória em C

```c
private volatile bool pronto = false;
private int dados = 0;

// Thread produtor
Task.Run(() => {
    dados = 42;
    Interlocked.MemoryBarrier(); // Barreira de memória explícita
    pronto = true;
});

// Thread consumidor
Task.Run(() => {
    while (!pronto) {
        Thread.Yield();
    }
    Interlocked.MemoryBarrier();
    int valor = dados;
});
```

6.2. Multithreading

Threads em C++

```cpp
// Criando uma thread
std::thread t([]() {
    std::cout << "Thread executando" << std::endl;
});

// Esperando a thread terminar
t.join();
```

Threads em C

```csharp
// Criando uma thread
Thread t = new Thread(() => {
    Console.WriteLine("Thread executando");
});
t.Start();

// Esperando a thread terminar
t.Join();
```

6.3. Futures e Promises vs. Task

C++: Futures e Promises

```cpp
// Usando std::async
std::future<int> futuro = std::async(std::launch::async, []() {
    std::this_thread::sleep_for(std::chrono::seconds(1));
    return 42;
});

// Obtendo o resultado
int resultado = futuro.get(); // Bloqueia até o resultado estar disponível
```

C#: Task

```csharp
// Usando Task
Task<int> tarefa = Task.Run(() => {
    Thread.Sleep(1000);
    return 42;
});

// Obtendo o resultado
int resultado = tarefa.Result; // Bloqueia até o resultado estar disponível
```

6.4. Paralelismo

C++: Algoritmos Paralelos (C++17)

```cpp
std::vector<int> v(1000);
std::iota(v.begin(), v.end(), 0); // Preenche com 0, 1, 2, ...

// Processamento paralelo
std::for_each(std::execution::par, v.begin(), v.end(), [](int& x) {
    x = x * x;
});
```

C#: PLINQ e Parallel

```
var numeros = Enumerable.Range(0, 1000).ToList();

// Usando PLINQ
var quadrados = numeros.AsParallel().Select(x => x * x).ToList();

// Usando Parallel
Parallel.ForEach(numeros, x => {
    Console.WriteLine(x * x);
});
```

7. STL (Standard Template Library)

7.1. Containers

A STL oferece vários tipos de containers, semelhantes às coleções em C#.

Sequence Containers

```
// Vector (similar a List<T> em C#)
std::vector<int> vec = {1, 2, 3, 4, 5};
vec.push_back(6);

// Deque (similar a LinkedList<T> em C#, mas com acesso aleatório)
std::deque<int> dq = {1, 2, 3};
dq.push_front(0);
dq.push_back(4);

// List (lista duplamente ligada)
std::list<int> lst = {1, 2, 3};
lst.push_back(4);
lst.push_front(0);
```

Associative Containers

```
// Map (similar a Dictionary<K,V> em C#)
std::map<std::string, int> idades;
idades["Alice"] = 30;
idades["Bob"] = 25;

// Set (similar a HashSet<T> em C#)
std::set<int> numeros = {1, 2, 3, 4, 5};
numeros.insert(6);
```

Unordered Containers

```
// Unordered map (hash table)
std::unordered_map<std::string, int> hash_idades;
hash_idades["Alice"] = 30;

// Unordered set (hash set)
std::unordered_set<int> hash_numeros = {1, 2, 3, 4, 5};
```

7.2. Algoritmos

A STL inclui muitos algoritmos genéricos, semelhantes ao LINQ em C#.

```
std::vector<int> v = {5, 2, 8, 1, 9};

// Ordenação
std::sort(v.begin(), v.end());

// Busca
auto it = std::find(v.begin(), v.end(), 8);

// Transformação
std::vector<int> quadrados(v.size());
std::transform(v.begin(), v.end(), quadrados.begin(), [](int x) { return x * x; });

// Redução
int soma = std::accumulate(v.begin(), v.end(), 0);
```

Equivalente em C# usando LINQ:

```
List<int> lista = new List<int> { 5, 2, 8, 1, 9 };

// Ordenação
var ordenada = lista.OrderBy(x => x).ToList();

// Busca
var elemento = lista.FirstOrDefault(x => x == 8);

// Transformação
var quadrados = lista.Select(x => x * x).ToList();

// Redução
int soma = lista.Sum();
```

8. Recursos Modernos de C++

8.1. Módulos (C++20)

Módulos são uma alternativa moderna ao sistema de headers em C++.

```
// Definição de módulo (math.cpp)
export module math;

export int add(int a, int b) {
    return a + b;
}

// Uso de módulo (main.cpp)
import math;

int main() {
    int result = add(2, 3);
    return 0;
}
```

8.2. Coroutines (C++20)

Coroutines em C++20 são semelhantes a async/await em C#.

```cpp
#include <coroutine>
#include <iostream>

struct Task {
    struct promise_type {
        Task get_return_object() { return {}; }
        std::suspend_never initial_suspend() { return {}; }
        std::suspend_never final_suspend() noexcept { return {}; }
        void return_void() {}
        void unhandled_exception() {}
    };
};

Task exemplo() {
    std::cout << "Início" << std::endl;
    co_await std::suspend_always{};
    std::cout << "Meio" << std::endl;
    co_await std::suspend_always{};
    std::cout << "Fim" << std::endl;
}
```

Equivalente em C#:

```csharp
async Task ExemploAsync()
{
    Console.WriteLine("Início");
    await Task.Yield();
    Console.WriteLine("Meio");
    await Task.Yield();
    Console.WriteLine("Fim");
}
```

9. Melhores Práticas

9.1. Idiomas de C++

RAII (Resource Acquisition Is Initialization) RAII é um padrão fundamental em C++ onde recursos são adquiridos durante a inicialização e liberados durante a destruição.

```cpp
class File {
private:
    FILE* handle;

public:
    File(const char* filename, const char* mode) {
        handle = fopen(filename, mode);
        if (!handle) throw std::runtime_error("Failed to open file");
    }

    ~File() {
        if (handle) fclose(handle);
    }
```

```
    // Proibir cópia
    File(const File&) = delete;
    File& operator=(const File&) = delete;

    // Permitir movimento
    File(File&& other) noexcept : handle(other.handle) {
        other.handle = nullptr;
    }

    File& operator=(File&& other) noexcept {
        if (this != &other) {
            if (handle) fclose(handle);
            handle = other.handle;
            other.handle = nullptr;
        }
        return *this;
    }

    // Operações no arquivo...
};
```

Rule of Three/Five/Zero

- **Rule of Three**: Se você precisar definir um destrutor, operador de cópia ou construtor de cópia, provavelmente precisará definir os três.
- **Rule of Five**: Com C++11, adicione construtor de movimento e operador de atribuição por movimento.
- **Rule of Zero**: Projete suas classes para não precisar de nenhum dos cinco métodos especiais.

pImpl (Pointer to Implementation) O idioma pImpl é usado para esconder detalhes de implementação e reduzir dependências.

```
// Header
class Widget {
private:
    class Impl;
    std::unique_ptr<Impl> pImpl;

public:
    Widget();
    ~Widget();
    void doSomething();
};

// Implementation
class Widget::Impl {
public:
    void doSomething() {
        // Implementação real
    }
};
```

```cpp
Widget::Widget() : pImpl(std::make_unique<Impl>()) {}
Widget::~Widget() = default;

void Widget::doSomething() {
    pImpl->doSomething();
}
```

9.2. Otimização e Performance

Evite Cópias Desnecessárias

```cpp
// Ruim: Cópia desnecessária
void processar(std::vector<int> vec) {
    // Processa vec
}

// Melhor: Referência const para evitar cópia
void processar(const std::vector<int>& vec) {
    // Processa vec sem modificá-la
}

// Para modificar: Referência não-const
void processar_e_modificar(std::vector<int>& vec) {
    // Modifica vec
}

// Para transferir propriedade: Referência rvalue
void processar_e_mover(std::vector<int>&& vec) {
    // Toma posse de vec
}
```

Use Algoritmos da STL Os algoritmos da STL são otimizados e geralmente mais eficientes que loops manuais.

```cpp
// Menos eficiente
int soma = 0;
for (const auto& valor : vetor) {
    soma += valor;
}

// Mais eficiente
int soma = std::accumulate(vetor.begin(), vetor.end(), 0);
```

Reserve Memória Antecipadamente

```cpp
// Sem reserva (múltiplas realocações)
std::vector<int> v;
for (int i = 0; i < 10000; ++i) {
    v.push_back(i);
}

// Com reserva (uma única alocação)
std::vector<int> v;
v.reserve(10000);
for (int i = 0; i < 10000; ++i) {
```

```
    v.push_back(i);
}
```

9.3. Guidelines

C++ Core Guidelines As C++ Core Guidelines são um conjunto de regras e recomendações para escrever código C++ moderno, seguro e eficiente.

Algumas regras importantes:

- **F.15**: Prefira funções constexpr para computação em tempo de compilação
- **C.20**: Se você pode evitar definir qualquer um dos métodos especiais, faça-o
- **R.1**: Gerencie recursos automaticamente usando RAII
- **ES.23**: Prefira escopo de variável o mais restrito possível
- **CP.20**: Use RAII, nunca chame mutex.lock e mutex.unlock diretamente

Google C++ Style Guide O Google C++ Style Guide fornece diretrizes para escrever código C++ limpo e consistente.

Algumas recomendações:

- Use nomes descritivos em camelCase para variáveis e funções
- Use nomes em PascalCase para tipos (classes, structs, enums)
- Evite macros, prefira constexpr e templates
- Limite o comprimento das linhas a 80 caracteres
- Use 2 espaços para indentação (não tabs)

Microsoft C++ Coding Conventions As Microsoft C++ Coding Conventions fornecem diretrizes para escrever código C++ no estilo Microsoft.

Algumas recomendações:

- Use nomes descritivos em camelCase para variáveis e PascalCase para funções e tipos
- Use 4 espaços para indentação
- Coloque chaves em linhas separadas
- Prefira tipos da STL a implementações personalizadas
- Use comentários para documentar o “porquê”, não o “como”

10. Ferramentas e Ecossistema

10.1. Compiladores

Os principais compiladores C++ são:

- **GCC**: Compilador GNU, disponível em sistemas Unix/Linux
- **Clang**: Parte do projeto LLVM, conhecido por mensagens de erro claras
- **MSVC**: Microsoft Visual C++, otimizado para Windows

10.2. Build Systems

- **CMake**: Sistema de build multiplataforma
- **Make**: Sistema de build tradicional em Unix
- **MSBuild**: Sistema de build da Microsoft

10.3. Debugging e Profiling

- **GDB**: GNU Debugger para sistemas Unix/Linux
- **LLDB**: Debugger do projeto LLVM
- **Visual Studio Debugger**: Integrado ao Visual Studio
- **Valgrind**: Ferramenta para detecção de vazamentos de memória
- **perf**: Ferramenta de profiling para Linux

Conclusão

C++ e C# são linguagens poderosas com diferentes filosofias e casos de uso. C++ oferece controle de baixo nível, eficiência e flexibilidade, enquanto C# prioriza produtividade, segurança e integração com o ecossistema .NET.

Como programador C# experiente, você encontrará em C++ uma linguagem que exige mais atenção aos detalhes, especialmente no gerenciamento de memória, mas que oferece possibilidades de otimização e controle que não estão disponíveis em C#.

Os conceitos aprendidos neste tutorial fornecem uma base sólida para explorar C++ em profundidade e aplicá-lo em projetos que exigem alto desempenho, interação direta com hardware ou integração com sistemas legados.

Recursos Adicionais

- cppreference.com - Documentação completa de C++
- C++ Core Guidelines
- Effective Modern C++ por Scott Meyers
- C++ Templates: The Complete Guide por David Vandevoorde e Nicolai M. Josuttis
- C++ Concurrency in Action por Anthony Williams