UFC

Resumo Cpp

Resumo C++

Henricky Monteiro
HenrickyL

Mar 2025

Fundamentos do C++ Moderno

Boas Práticas e Filosofias de Código

A Regra 80/20 (Princípio de Pareto)

- **80% dos resultados vêm de 20% dos esforços**
- Em programação:
  - 20% das ferramentas e conceitos resolvem 80% dos problemas
  - Foque no essencial primeiro: loops, funções, structs/classes, vetores
- Mantenha isso em mente antes de buscar "otimizações prematuras" ou recursos avançados

Código é para Humanos

- **O compilador entende qualquer coisa — as pessoas não**
- O código deve ser:
  - **Legível** (nomes claros, estrutura limpa)
  - **Consistente** (padrões de indentação, nomes, organização)
  - **Comentado** quando necessário — mas o ideal é que o código "fale por si"
- Um bom código é fácil de entender e manter, não só de executar

Código Limpo (Clean Code)

- Dê nomes que façam sentido
- Evite duplicação
- Funções pequenas e com propósito claro
- Use espaços, quebre em linhas, organize bem

```cpp
// ruim
void f(int x, int y) { if(x>y){x++;}else{y--;} }

// melhor
void ajustarValores(int& maior, int& menor) {
    if (maior > menor) {
        maior++;
    } else {
        menor--;
    }
}
```

Software é Coletivo

- **Projetos reais de software são feitos por equipes**
- Mesmo em pequenos projetos:
  - Há prazo
  - O código passa por várias pessoas
  - Precisa evoluir com o tempo
- Por isso usamos:
  - **Arquitetura de software** (separar em módulos, camadas, responsabilidades)
  - **Controle de versão** (`Git` para salvar, compartilhar e colaborar no código)

Reflexões Finais

- Programar é resolver problemas de forma clara
- Código não é arte abstrata — é uma ferramenta de comunicação
- Bons hábitos desde o começo ajudam muito a longo prazo
- "*Qualquer tolo pode escrever código que um computador entende. Bons programadores escrevem código que humanos conseguem entender.*" — Martin Fowler

C++ Básico

Função `main`

- Todo programa em C++ começa pela função `main`
- Ela retorna um `int` e deve retornar `0` se o programa terminar corretamente

```cpp
#include <iostream>

int main() {
    std::cout << "Olá, mundo!" << std::endl;
    return 0;
}
```

- **Inclusão de bibliotecas:**
  - `<...>` inclui bibliotecas padrão (ex: `<iostream>`, `<vector>`, `<string>`, `<memory>`)
  - `"..."` é usado para incluir arquivos locais, como `"utils.h"`

```cpp
#include <iostream>   // entrada e saída padrão
#include <vector>     // estrutura de vetor dinâmica
#include <string>     // manipulação de strings
#include <memory>     // smart pointers como unique_ptr
#include "meuarquivo.h" // arquivo local do projeto
```

Compilação de Código em C++

No Linux/macOS:

- Compilação simples com `g++`:

```
g++ main.cpp tree.cpp -std=c++17 -Wall -o programa
./programa
```

- Ferramentas úteis:
  - `make` — Automação da compilação com Makefile
  - `cmake` — Geração multiplataforma de builds
  - `valgrind` — Análise de memória
  - `clang-tidy` — Análise estática de código

Compilação de Código em C++

No Windows:

- Usando `MinGW` no terminal (Git Bash, CMD ou PowerShell):

```
g++ main.cpp tree.cpp -std=c++17 -Wall -o programa.exe
.\programa.exe
```

- Alternativas:
  - **MSVC** (Visual Studio) — via `.sln` ou Developer Prompt
  - **WSL** (Windows Subsystem for Linux) — usar comandos de Linux
  - `make` — Automação da compilação com Makefile
- **Chocolatey** - Gerenciador de pacotes windows

```
>choco install mingw -y

>choco install make -y
```

Entrada e Saída

- Para imprimir no console: `std::cout`
- Para ler do teclado: `std::cin`

```cpp
#include <iostream>

int main() {
    int idade;
    std::cout << "Digite sua idade: ";
    std::cin >> idade;
    std::cout << "Você tem " << idade << " anos." << std::endl;
}
```

Tipos de Dados

- Tipos primitivos mais comuns:
  - `int` — inteiros
  - `float`, `double` — números reais
  - `char` — caracteres
  - `bool` — booleanos
- Modificadores: `unsigned`, `short`, `long`

Conversão de Tipos e Modificadores

- Conversões explícitas ajudam a evitar comportamentos inesperados.
- **Modificadores comuns:** `U` (unsigned), `L` (long)
- **Conversões seguras:** `static_cast`, `dynamic_cast`
- **Evite:** `reinterpret_cast` e `const_cast`, a menos que saiba o que está fazendo

```cpp
int x = 10;
double y = static_cast<double>(x); // conversão segura entre tipos numéricos

unsigned int a = 10U; // 'U' indica unsigned
long b = 1000L;        // 'L' indica long
short c = 100;

class Base { virtual void foo() {} };
class Derived : public Base {
    void foo() override {}
};

Base* bptr = new Derived();
Derived* dptr = dynamic_cast<Derived*>(bptr); //conversão em tempo de execução
if (dptr != nullptr)
    std::cout << "Conversão bem-sucedida." << std::endl;
```

Estruturas de Decisão

- Permitem executar caminhos diferentes com base em condições
- `if`: mais flexível, permite comparações complexas

```cpp
int x = 10;
if (x > 0) {
    std::cout << "Positivo\n";
} else if (x < 0) {
    std::cout << "Negativo\n";
} else {
    std::cout << "Zero\n";
}
```

- `switch/case`: útil para comparar uma variável contra múltiplos valores fixos

```cpp
char opcao = 'B';
switch (opcao) {
    case 'A': std::cout << "Selecionou A\n"; break;
    case 'B': std::cout << "Selecionou B\n"; break;
    default: std::cout << "Outro\n"; break;
}
```

Laços de Repetição

- Reduzir repetição de código

```cpp
// for: número conhecido de repetições
for (int i = 0; i < 5; i++) {
    std::cout << i << std::endl;
}
```

```cpp
// while: repete até a condição ser falsa
int i = 0;
while (i < 5) {
    std::cout << i << std::endl;
    i++;
}
```

- Existe o `do while` também

Ponteiros

- Um **ponteiro** armazena o endereço de memória de uma variável
- Operadores:
- Podem ser nulos, precisam ser desreferenciados
  - `&` — endereço
  - `*` — desreferenciação

```cpp
int a = 42;
int* p = &a;

std::cout << "Valor de a: " << *p << std::endl;
std::cout << "Endereço de a: " << &a << std::endl;
```

```cpp
Node node = new Node(10);

// -> equivale a desreferenciação (*node).key
std::cout << "Chave do Nó : " << node->key << std::endl;
```

Funções

- Usamos **funções** para organizar o código e evitar repetições
- **Sintaxe:** tipo de retorno, nome, parâmetros e bloco de código

```cpp
int soma(int a, int b) {
    return a + b;
}
```

```cpp
// Função com valor padrão
// Eleva um número a uma potência (default é 2)
int potencia(int base, int expoente = 2) {
    int resultado = 1;
    for (int i = 0; i < expoente; ++i)
        resultado *= base;
    return resultado;
}
```

Função com Retorno por Referência

- Usar `&` permite modificar variáveis externas diretamente
- Útil quando queremos **retornar múltiplos valores** sem usar struct ou tuple

```cpp
// Função que calcula soma e produto de dois números
void calcular(int a, int b, int& soma, int& produto) {
    soma = a + b;
    produto = a * b;
}
int main() {
    int x = 4, y = 5;
    int s = 0, p = 0;

    calcular(x, y, s, p); // s e p são modificados pela função

    std::cout << "Soma: " << s << ", Produto: " << p << std::endl;
}
```

- Boa prática para desempenho e múltiplos retornos

Vetores em C++

- **Vetores** são arrays de elementos do mesmo tipo
- Devem ter tamanho fixo conhecido ou usar alocação dinâmica

```cpp
// Array estático
int valores[5] = {1, 2, 3, 4, 5};
```

```cpp
// Array dinâmico com ponteiro
int* v = new int[5];
v[0] = 10;
delete[] v;
```

Redimensionamento Manual (Alocação Dinâmica)

- Vetores estáticos têm tamanho fixo — para mudar, aloque novo e copie os dados
- Útil para economizar memória (ex: cortar tamanho pela metade)

```cpp
int* redimensionar(int* v, int tamanhoAtual, int novoTamanho) {
    int* novo = new int[novoTamanho];
    for (int i = 0; i < novoTamanho; ++i)
        novo[i] = v[i]; // copia elementos antigos
    delete[] v; // libera o antigo
    return novo; // devolve novo ponteiro
}

int main() {
    int* v = new int[6]{1,2,3,4,5,6};
    int tamanho = 6;

    // reduz tamanho para metade
    tamanho /= 2;
    v = redimensionar(v, 6, tamanho);

    for (int i = 0; i < tamanho; ++i)
        std::cout << v[i] << " "; // 1 2 3

    delete[] v;
}
```

- **Cuidado:** sempre liberar memória antiga com `delete[]`

Erros Comuns em C++

- **Divisão por zero:** causa comportamento indefinido.

```
int x = 10, y = 0;
int resultado = x / y; // ERRO: divisão por zero
```

- **Segmentation fault:** acesso inválido à memória.

```
int* ptr = nullptr;
*ptr = 42; // ERRO: ponteiro nulo (null pointer)
```

```
Node* node = nullptr;
node->key = 42; // ERRO: ponteiro nulo (null pointer)
```

```
int arr[3] = {1, 2, 3};
int x = arr[5]; // ERRO: acesso fora do array (out of range)
```

- Evite esses erros usando:
  - Verificação de divisores antes de dividir
  - Inicialização adequada de ponteiros

Tipo Abstrato de Dados

Organização de Arquivos em C++

- Separar o código em `.h` (declarações) e `.cpp` (implementações)
- Usar `#pragma once` ou `#ifndef / #define` para evitar inclusões múltiplas

```cpp
// arquivo: tree.h
#ifndef QXD_TREE
#define QXD_TREE
class Tree {
public:
    void insert(int value);
};
#endif
```

```cpp
// arquivo: tree.cpp
#include "tree.h"
#include <iostream>

void Tree::insert(int value) {
    std::cout << "Inserting: " << value << std::endl;
}
```

Struct vs Class

- `struct` tem membros públicos por padrão; `class`, privados
- Use `class` para encapsulamento e métodos (Orientação a Objetos)

```cpp
struct Node {
     int value;
     Node* left;
     Node* right;
};
```

```cpp
class Tree {
private: // atributos
    Node* _root;
public: // métodos públicos
    Tree();//construtor
    void insert(int value);
    void clear();
    int countNodes() const;
    void print() const;
private: // métodos privados
    void print(const Node* node) const;
    //...
};
```

Tipos Abstratos de Dados (TAD)

- A interface esconde a implementação interna

```
//stack.h
class Stack {
private:
    int* data;
    int top;
    int capacity;
public:
    Stack(int cap);
    ~Stack();
    void push(int value);
    int pop();
};
```

- Usados para inicializar e limpar objetos

```
//stack.cpp
Stack::Stack(int cap) : capacity(cap), top(-1) {
    data = new int[cap];
}

Stack::~Stack() {
    delete[] data;
}
```

Orientação a Objetos (OO)

UFC

Encapsulamento

- Esconde os detalhes internos de implementação
- Controla o acesso via métodos públicos
- Boa prática: usar `public`, `private` e `protected`

```cpp
class Conta {
private:
    double _saldo;
public:
    Conta() : _saldo(0) {}
    void depositar(double valor) {
        if(_eValido(valor))
            saldo += valor;
    }
    double getSaldo() const { return saldo; }
private:
    bool _eValido(double valor);
};
```

Sobrecarga de Métodos e Operadores

- Permite múltiplas versões de uma método
- Operadores também podem ser reimplementados

```
class Vetor {
public:
    int x, y;
    Vetor(int a, int b) : x(a), y(b) {}
    Vetor operator+(const Vetor& v) const {
        return Vetor(x + v.x, y + v.y);
    }
};
```

Construtores e Destrutores

- Construtores inicializam objetos ao serem criados
- Destrutores liberam recursos ao final da vida do objeto

```cpp
class Exemplo {
public:
    Exemplo() { std::cout << "Construtor\n"; }
    ~Exemplo() { std::cout << "Destrutor\n"; }
};
```

Os 4 Pilares da OO

- **Abstração**: foco no que importa, esconde complexidade
- **Encapsulamento**: protege dados internos
- **Herança**: reaproveita e especializa código
- **Polimorfismo**: múltiplas formas de um mesmo método

```cpp
class Brinquedo {
public:
    virtual void acao() const = 0;
    virtual ~Brinquedo() = default;
};

class Carrinho : public Brinquedo {
public:
    void acao() const override {
        std::cout << "Carrinho andando!\n";
    }
};

class Boneca : public Brinquedo {
public:
    void acao() const override {
        std::cout << "Boneca falando!\n";
    }
};
```

OO em Ação

- Podemos usar um vetor de ponteiros para a classe base:

```cpp
#include <vector>

int main() {
   std::vector<Brinquedo*> brinquedos;

    brinquedos.push_back(new Carrinho());
    brinquedos.push_back(new Boneca());

    for (Brinquedo* b : brinquedos) {
        b->acao(); // Executa método apropriado
    }

    // Libera memória alocada
    for (Brinquedo* b : brinquedos) {
        delete b;
    }
    return 0;
}
```

- Reaproveitar e estender código de uma classe base: isso é **Herança**!
- Mesmo tipo, diferentes comportamentos: isso é **Polimorfismo**!

Intermediário e Estruturas Padrão

Orientação a Objetos Intermediário

Métodos `virtual`

- O modificador `virtual` permite polimorfismo em tempo de execução
- O método mais derivado é chamado, mesmo com ponteiro para classe base
- Use `override` para indicar substituições

```cpp
class Base {
public:
    virtual void f() const { std::cout << "Base\n"; }
};
class Derivada : public Base {
public:
    void f() const override { std::cout << "Derivada\n"; }
};
```

Membros `static`

- Membros `static` pertencem à **classe** e não às instâncias
- Usados para:
  - Compartilhar dados comuns entre objetos (ex: contador global)
  - Criar métodos utilitários (ex: operações matemáticas, fábricas)
  - Implementar padrões como `Singleton`, `Factory`, etc.

```
class Matriz {
public:
    static int instancias;

    Matriz() { instancias++; }

    static Matriz somar(const Matriz& a, const Matriz& b) {
        // retorna nova matriz com soma dos elementos
        return Matriz(); // simplificado
    }
};
int Matriz::instancias = 0;

int main() {
    Matriz m1, m2;
    Matriz m3 = Matriz::somar(m1, m2);
    std::cout << "Instâncias: " << Matriz::instancias << std::endl;
}
```

Interfaces em C++

- **Interface** é simulada com classes abstratas puras
- Todos os métodos são `virtual` e com `= 0`
- Não possui implementação — apenas contrato

```cpp
class Imprimivel {
public:
    virtual void imprimir() const = 0;
    virtual ~Imprimivel() = default;
};

class Produto : public Imprimivel {
public:
    void imprimir() const override {
        std::cout << "Produto\n";
    }
};
```

Herança Múltipla

- Uma classe pode herdar de múltiplas bases
- Pode gerar ambiguidade — resolvida com escopo ou herança virtual
- Use com cuidado

```cpp
class A { public: void f() {} };
class B { public: void g() {} };
class C : public A, public B {};

C obj;
obj.f(); obj.g();
```

Conceitos Intermediários

Recursão

- Uma função recursiva chama a si mesma para resolver subproblemas
- Toda recursão precisa de um caso base

```cpp
int fatorial(int n) {
    if (n == 0) return 1;      // caso base
    return n * fatorial(n - 1); // chamada recursiva
}
```

```cpp
void decrescente(int n){
    if(n>=0){// chamada recursiva
        cout << n << ' ';
        decrescente(n-1);
    }// caso base
}
```

- A pilha de chamadas resolve o problema de trás para frente

Recursão

- A recursão pode ser usada para modificar estruturas in-place
- Neste exemplo, os extremos são trocados até se encontrarem

```
void reverte(int* v, int n, int i = 0) {
    if (n <= i) {
        std::cout << "*\n";  // caso base: índices se cruzaram
    } else {
        std::swap(v[i], v[n - 1]); // troca extremos
        reverte(v, n - 1, i + 1);  // avança para centro
    }
}

int main() {
    int v[] = {1, 2, 3, 4, 5, 6};
    reverte(v, 6);
    // v agora é: {6, 5, 4, 3, 2, 1}
}
```

- Técnica comum em problemas de divisão e conquista

Ponteiros vs Referências

- **Ponteiros** `T*` armazenam endereços de memória e podem ser nulos
- **Referências** `T&` são apelidos para variáveis existentes e não podem ser nulas

```cpp
void modify(Node& ref) {
    node.key += 10; // altera valor diretamente
}

void modifyNode(Node* ptr) {
    if (node) node->key += 10; // necessário verificar ponteiro
}
```

- Modificadores com `const`:
  - `const float& x`: referência constante, evita cópia
  - `float*`: ponteiro para float
  - `const float*`: ponteiro para float constante
  - `float* const`: ponteiro constante para float
  - `const float* const`: ponteiro constante para float constante

Constantes com `const`

- Usar `const` para garantir imutabilidade
- Bom para proteger variáveis de alterações acidentais

```
void BinarySearchTree::print() const{
    _root = nullptr;
    _print(_root);
}
/*error: assignment of member 'BinarySearchTree::_root' in read-only object
  111 |     _root = nullptr;
      |     ~~~~~~^~~~~~~~~ */
```

```
void BinarySearchTree::_print(const Node* node) const{
    node->left = nullptr; // <----
    if(node != nullptr){
        std::cout <<node->key << " " ;
        _print(node->left);
        _print(node->right);
    }else
        std::cout<< " #";
}
/*error: assignment of member 'Node::left' in read-only object
  115 |     node->left = nullptr;
      |     ~~~~~~~~~~~^~~~~~~~~ */
```

const no Retorno da Função

- Retornar uma **referência constante** impede alterações no valor retornado
- Protege membros internos ao expor dados sem permitir modificação
- Exemplo:

```cpp
class Produto {
private:
    std::string nome;
public:
    // Permite leitura, mas impede modificação externa
    const std::string& getNome() const {
        return nome;
    }
};

int main() {
    Produto p;
    // getNome() retorna const ref - não pode ser modificado:
    // p.getNome() = "NovoNome"; // ERRO!
    std::cout << p.getNome() << std::endl;
}
```

- Evita cópia (eficiente) e garante segurança (imutável)

Alocação Dinâmica

- **Manual:**
  - `new` aloca memória no heap
  - `delete` libera a memória
  - Risco: vazamento de memória e `segmentation fault`
- **Preferencial (C++ moderno):**
  - `std::unique_ptr` — propriedade única, liberação automática
  - `std::shared_ptr` — ponteiro com referência compartilhada
  - Criados com `std::make_unique` ou `std::make_shared`

```cpp
#include <iostream>
#include <memory>

int* p = new int(10); // alocação manual
std::cout << *p << std::endl;
delete p; // liberação manual

// C++ moderno
auto up = std::make_unique<int>(99); // unique_ptr
auto sp = std::make_shared<int>(42); // shared_ptr
std::cout << *up << " " << *sp << std::endl;
```

STL: Estruturas de Dados Comuns

- A STL (*Standard Template Library*) oferece estruturas de dados prontos para uso:
  - **std::vector:** vetor dinâmico, permite acesso diretamente pelo índice
  - **std::stack:** pilha (LIFO), usa `push`, `pop`, `top`
  - **std::queue:** fila (FIFO), usa `push`, `pop`, `front`, `back`
  - **Iteradores:** percorrem estruturas
  - **std::list:** lista duplamente encadeada (sem acesso por índice direto)

```cpp
#include <vector>
#include <stack>
#include <queue>

std::vector<int> v = {1, 2, 3};
v.push_back(4);

std::stack<int> s;
s.push(1);
s.pop();

std::queue<int> q;
q.push(10);
q.pop();

for (auto it = v.begin(); it != v.end(); ++it) //iterator
{    std::cout << *it << std::endl; }
for (auto x : v)// for each
{    std::cout << x << std::endl; }
```

C++ Intermediários

Enumerações

- Permite representar conjuntos de constantes nomeadas.
- Dois tipos: `enum` tradicional e `enum class` (C++11).

```cpp
enum Dia { Segunda, Terca, Quarta };
enum class Cor { Vermelho, Azul, Verde };

Dia d = Segunda;
Cor c = Cor::Azul;
```

```cpp
enum class Direcao { N,NE,E,SE,S,SW,W,NW };

for (int d = (int)Direcao::N; d <= (int)Direcao::NW; d++) {
    std::cout << (Direcao)d << std::endl;
}
```

Sobrecarga de Operadores

- Permite definir comportamento para operadores com objetos.
- Ex: somar dois vetores, comparar objetos, etc.

```cpp
class Vetor {
  int _x; int _y;
public:
  Vetor(int x, int y) : _x(x), _y(y) {}
  Vetor operator+(const Vetor& outro) {
    return Vetor(_x + outro._x, _y+outro.y);
  }
};
```

```cpp
Vector c = Vector(1,3) + Vector(2,-1);
```

Templates

- `template<typename ...>` permite criar código genérico.
- Usado para funções e classes com tipos `T` variáveis.

```
template <typename T>
T soma(T a, T b) {
  return a + b;
}

int x = soma(2, 3);          // int
float y = soma(1.5f, 2.5f); // float
```

```
template <typename T>
struct Node {
  int key;
  T value;
  Node<T>* left;
  Node<T>* right;

  Node(int k, T v, Node<T>* l = nullptr, Node<T>* r = nullptr)//...
}

Node<int> n(10, 25);
Node<std::string> n2(3,"abc")
```

Arrays e Strings

Resumo Cpp

Fundamentos
Boas Práticas e Filosofias
Cpp Básico
TAD
POO

Conceitos Intermediários
POO Plus
Conceitos
Cpp Intermediário
Extra

- Arrays são blocos de memória contínuos
- Strings em C++ podem ser manipuladas com `std::string` ou com arrays de `char`

```
char nome[10] = "Ana";
std::string sobrenome = "Silva";
```

```
#include <string>

std::string s = "Hello";
s += " World"; // concatenação de strings
std::cout << s.substr(1, 4) << std::endl; // imprime "ello"
std::cout << s.find("World") << std::endl; // imprime 6 - índice onde "World" começa

s.length();        // tamanho da string (int)
s.empty();         // verifica se está vazia (bool)
s.replace(0, 5, "Hi"); // substitui os 5 primeiros carac. por "Hi" → s vira "Hi World"
s.erase(2, 3);         // remove 3 carac. após o índice 2 → "Hi World" vira "Hiorld"
s.insert(1, "abc");    // insere "abc" na posição 1 → vira "Habciorld"
s.append("!!!");       // adiciona "!!!" no final da string → "Habciorld!!!"
s.compare("Hi") == 0;  // compara com "Hi", retorna 0 se forem iguais (int)

int n = std::stoi("123");          // converte string para int
double d = std::stod("3.14");      // converte string para double
std::string t = std::to_string(99); // converte número para string
```

Tratamento de Erros

- O C++ usa exceções para lidar com erros em tempo de execução
- Blocos `try/catch` capturam exceções lançadas com `throw`
- Ideal para erros inesperados ou críticos
- Tipos padrão de exceções: `std::runtime_error`, `std::logic_error`, entre outros

```cpp
#include <iostream>
#include <stdexcept> // std::runtime_error

void dividir(int a, int b) {
    if (b == 0)
        throw std::runtime_error("Divisão por zero");
    std::cout << "Resultado: " << a / b << std::endl;
}

int main() {
    try {
        dividir(10, 0);
    } catch (const std::runtime_error& e) {
        std::cerr << "Erro: " << e.what() << std::endl;
    }
}
```

Exceções Personalizadas

- Podemos criar nossas próprias classes de exceção
- Basta herdar de `std::exception` ou `std::runtime_error`
- Permite mensagens e tipos de erro específicos do domínio da aplicação

```cpp
#include <iostream>
#include <stdexcept>

class TreeError : public std::runtime_error {
public:
    TreeError(const std::string& msg) : std::runtime_error(msg) {}
};

void insert(int value) {
    if (value < 1)
        throw TreeError("Valor negativo não é permitido");
    std::cout << "Inserido: " << value << std::endl;
}

int main() {
    try {
        insert(-1);
    } catch (const TreeError& e) {
        std::cerr << "Erro na árvore: " << e.what() << std::endl;
    }
}
```

Expressões Lambda

- Funções anônimas criadas inline.
- Úteis com algoritmos e callbacks.

```
auto quadrado = [](int x) { return x * x; };
int r = quadrado(4); // 16
```

- Funções podem ser passadas como parâmetros.
- Pode-se usar ponteiros ou ‘std::function‘.

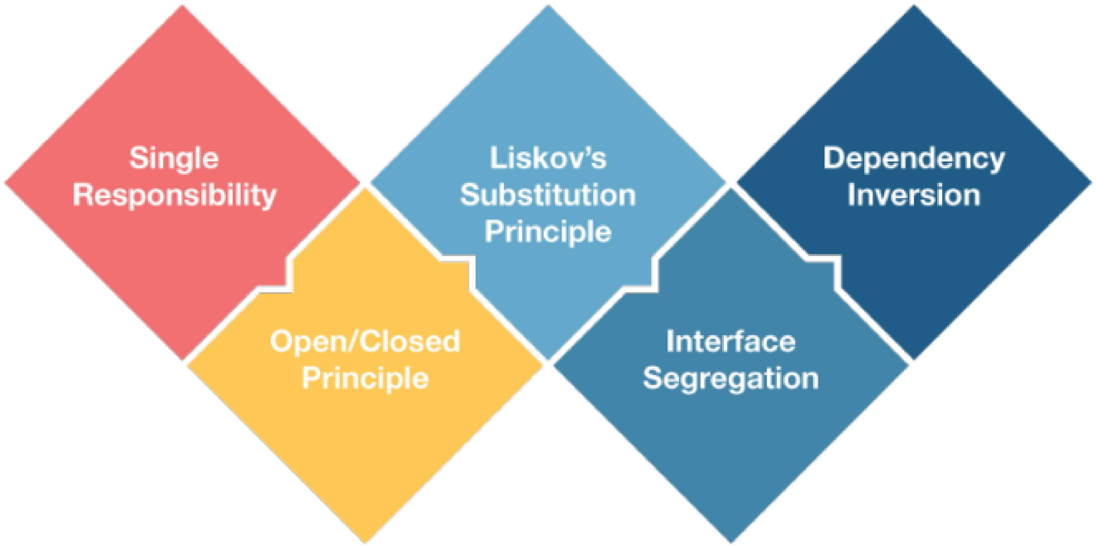
```
void aplicar(int x, int (*f)(int)) {
  std::cout << f(x);
}

int dobrar(int x) { return x * 2; }

aplicar(5, dobrar); // imprime 10
```

Boas práticas

Boas Práticas em C++ Moderno

- Use `nullptr` em vez de `NULL`
- Use `auto` quando o tipo for evidente
- Sempre inicialize suas variáveis
- Use `const` para argumentos que não devem ser modificados
- Prefira `std::unique_ptr` e `std::shared_ptr` a `new/delete`
- Use *range-based for* para iterar de forma segura
- Marque métodos que sobrescrevem com `override`
- Evite `using namespace std;` em headers
- Separe interface (header) e implementação (cpp)
- Escreva funções pequenas, com responsabilidade única

- **Princípios SOLID**: boas práticas para código orientado a objetos



- **Refatoração**: melhorar código sem mudar sua funcionalidade
- **Testes automatizados**: garantem confiança nas mudanças de código
- **Arquiteturas de software**: organização modular (ex: MVC, camadas)
- Esses conceitos ajudam a construir sistemas robustos e sustentáveis

Padrões de Projeto (Design Patterns)

- Soluções reutilizáveis para problemas comuns de design
- Catalogados em três grupos principais:
  - **Criacionais**: Singleton, Factory, Builder
  - **Estruturais**: Adapter, Decorator, Composite
  - **Comportamentais**: Observer, Strategy, Iterator
- Evitam reinvenção da roda e facilitam a comunicação entre devs
- *Primeiro use bastante algo pronto antes de tentar reinventar*
- Veja em www.refactoring.guru
- Exemplos:
  - **Iterator** — percorre coleções sem expor sua estrutura interna
  - **Singleton** — garante que uma classe tenha apenas uma instância e fornece um ponto global de acesso
  - **Observer** — um objeto notifica outros quando seu estado muda
  - **Strategy** — permite trocar algoritmos em tempo de execução
  - **Decorator** — adiciona responsabilidades a objetos dinamicamente, sem alterar seu código

Fim!