DOMINANDO C++ EM 1 ANO

Caderno de Respostas

Gabarito comentado: 59 aulas de prática, do #01 ao #59

C++23 · clang ou gcc · 10 módulos · 91 questões · 2 desafios integradores

Autor Desconhecido · theknowledgeisle.com

Consulte este gabarito só depois de tentar.

As respostas são comentadas para ensinar, não só para conferir. Nos exercícios de código existem outras soluções igualmente válidas: o que importa é entender por que a sua funciona e o que ela troca em relação à mostrada aqui.

Os programas com um quadro verde de **Saída** foram compilados com clang++ (-std=c++23 -Wall -Wextra) e libstdc++ 14, e executados no Linux, e a saída mostrada é a real.

Sumário

Módulo	Tema	Aulas	Questões
MÓDULO 1	O que muda ao sair do C	#01–#06	Q1–Q11
MÓDULO 2	Objetos e posse de recursos	#07–#12	Q12–Q22
MÓDULO 3	Herança, polimorfismo e operadores	#13–#16	Q23–Q32
MÓDULO 4	A biblioteca padrão: contêineres, iteradores e algoritmos	#17–#23	Q33–Q44
MÓDULO 5	Programação genérica	#24–#26	Q45–Q50
MÓDULO 6	Erros, valores opcionais e o C++ moderno	#27–#30	Q51–Q58
MÓDULO 7	Concorrência	#31–#34	Q59–Q65
MÓDULO 8	Ferramentas e qualidade	#35–#38	Q66–Q71
MÓDULO 9	Projeto final e extensões: o mini banco de dados, a web e a persistência	#39–#52	Q72–Q83
MÓDULO 10	Jogos	#53–#59	Q84–Q91
DESAFIOS	Desafios integradores	todas	D1–D2

MÓDULO 1

O que muda ao sair do C

Saída com std::print, referências, const, sobrecarga, std::string, nullptr e auto: o que o C++ muda para quem já domina C.

Artigos 01 a 06

#01 A Filosofia por Trás do C++ e a Saída Elegante do printf
#02 Referências e Ponteiros, Frente a Frente
#03 A Disciplina do const — Promessas que o Compilador Cobra
#04 Um Nome, Vários Sentidos — Sobrecarga de Funções
#05 O Fim do char* — Textos de Verdade com std::string
#06 Adeus NULL, Bem-vindo à Dedução — nullptr e auto

Perguntas conceituais — respostas

Q1 Por que não printf

> *O `printf` funciona em C++. Que dois problemas dele o `std::print` (C++23) e o `std::format` resolvem? Dê um exemplo de erro que o `printf` deixa passar e o `std::print` não.*

O `printf` **não conhece os tipos** dos argumentos: confia na string de formato. Os problemas:

- **Tipo errado passa despercebido**: `printf("%d", 3.5)` compila (no máximo com aviso) e imprime lixo, porque o `double` é lido como `int`. Com `std::print("{}", 3.5)`, o tipo vem do próprio argumento, e um formato incompatível como `{:d}` para um `double` é **erro de compilação**, já que a string de formato é verificada em tempo de compilação.
- **Não é extensível**: não há como ensinar o `printf` a imprimir um tipo seu. O `std::format` aceita tipos do usuário com uma especialização de `std::formatter`.

Bônus: o número de argumentos também é conferido, e `{}` dispensa decorar `%ld`, `%zu` e `%lld`.

Q2 Referência ou ponteiro

> *Cite três diferenças entre uma referência (`int&`) e um ponteiro (`int*`). Quando o ponteiro continua sendo a escolha certa?*

- A referência **precisa ser inicializada** e não existe referência "vazia"; o ponteiro pode ser `nullptr`.
- A referência **não pode ser reapontada**: `r = y` copia o valor de `y` para o objeto referido. O ponteiro pode mudar de alvo.
- A referência é usada **com a sintaxe do próprio objeto** (sem `*` nem `->`), e não existe aritmética de referência.

O ponteiro continua certo quando o "nada" é um valor legítimo (parâmetro opcional, fim de lista encadeada), quando o alvo precisa mudar ao longo da vida da variável e na interface com APIs em C. Para "talvez um valor", o C++ moderno também oferece o `std::optional` (módulo 6).

Q3 const em quatro lugares

Explique o que cada `const` promete em `const char* const nome(const Pessoa& p) const;`, supondo que a função seja membro de uma classe.

Lendo da esquerda para a direita:

1. `const char*`: os caracteres apontados pelo retorno não podem ser alterados por quem recebe.
2. `* const`: o ponteiro devolvido é constante. Num valor de retorno esse `const` não protege nada útil (quem recebe copia o ponteiro) e os compiladores costumam avisar; é ruído.
3. `const Pessoa& p`: a função recebe a pessoa por referência, sem cópia, e promete não alterá-la.
4. O `const` final: a função membro não altera o objeto em que é chamada (`this` é um ponteiro para `const`), e por isso pode ser chamada em objetos constantes.

O compilador **cobra** essas promessas: violá-las é erro de compilação, não convenção.

Q4 Sobrecarga e ambiguidade

Por que a chamada `f(0)` é ambígua quando existem `void f(long)` e `void f(double)`, e não é quando existe também `void f(int)`?

A resolução de sobrecarga procura a função que exige a conversão "mais barata". O literal `0` é `int`. Com `f(int)` há uma **correspondência exata**, que vence qualquer conversão. Sem ela, tanto `int → long` quanto `int → double` são **conversões** de mesmo nível (nenhuma é promoção), e o compilador não tem critério para escolher: é erro de ambiguidade. A saída é ser explícito: `f(0L)` ou `f(0.0)`.

Q5 Por que nullptr

Em C, `NULL` costuma ser `0` ou `((void*)0)`. Que problema isso cria com sobrecarga em C++, e como o `nullptr` o resolve?

Em C++ o `NULL` é tipicamente `0` (ou `0L`), um **inteiro**. Com `void g(int)` e `void g(char*)`, a chamada `g(NULL)` escolhe `g(int)`, o contrário do que se pretendia (ou fica ambígua, conforme a definição). O `nullptr` tem o tipo próprio `std::nullptr_t`, que converte para qualquer ponteiro mas **não** para inteiro: `g(nullptr)` chama `g(char*)` sem dúvida.

Q6 Quando auto atrapalha

O `auto` deduz o tipo pela inicialização. Que tipo tem `x` em cada linha, e qual delas provavelmente é um erro?

```
const std::string nome = "Ana";
auto a = nome;
auto& b = nome;
auto c = {1, 2, 3};
auto d = 5u - 10;
```

- `a` é `std::string`: o `auto` sozinho **descarta** `const` e referência, e `a` é uma cópia.

- `b` é `const std::string&`: com `auto&`, o `const` do original é preservado.
- `c` é `std::initializer_list<int>`, e não um vetor: surpresa clássica.
- `d` é `unsigned int` e vale 4294967291 (com `unsigned` de 32 bits): a conta mistura `unsigned` com `int`, o `int` é convertido e o resultado "negativo" dá a volta. É o provável erro; o `auto` só tornou o tipo menos visível.

Exercícios de código — respostas

Q7 Formatando uma tabela

> *Escreva um programa que imprima, com `std::println`, uma tabela de três produtos (nome, quantidade, preço) com o nome alinhado à esquerda em 12 colunas, a quantidade à direita em 4 e o preço com duas casas decimais à direita em 9. Na última linha, mostre o total.*

Os especificadores do `std::format` seguem `{:[alinhamento][largura][.precisão][tipo]}`: `<` à esquerda, `>` à direita.

```
#include <print>
#include <string>
#include <vector>

struct Item {
    std::string nome;
    int qtd;
    double preco;
};

int main() {
    std::vector<Item> itens{{"Caneta", 10, 2.5}, {"Caderno", 3, 18.9}, {"Mochila", 1, 149.0}};
    double total = 0;
    std::println("{:<12}{:>4}{:>9}", "Produto", "Qtd", "Preço");
    for (const auto& i : itens) {
        std::println("{:<12}{:>4}{:>9.2f}", i.nome, i.qtd, i.preco * i.qtd);
        total += i.preco * i.qtd;
    }
    std::println("{:<16}{:>9.2f}", "Total", total);
}
```

Saída

```
Produto      Qtd    Preço
Caneta        10    25.00
Caderno        3    56.70
Mochila        1   149.00
Total             230.70
```

Repare que "Preço" tem acento e mesmo assim ficou alinhado: o `std::format` mede a largura em caracteres Unicode quando a string de formato é UTF-8, coisa que o `printf("%-12s")` não faz (ele conta bytes).

Q8 Troca com referências

Escreva `void trocar(int& a, int& b)` e compare com a versão de C, que recebe ponteiros. Depois escreva `void dobrar_todos(std::vector<int>& v)` que dobre cada elemento usando um `for` com referência, e mostre o que acontece se você esquecer o `&` no laço.

```
#include <print>
#include <vector>

void trocar(int& a, int& b) {
    int t = a;
    a = b;
    b = t;
}

void dobrar_todos(std::vector<int>& v) {
    for (int& x : v)  // sem o &, x seria uma cópia e o vetor não mudaria
        x *= 2;
}

void dobrar_errado(std::vector<int>& v) {
    for (int x : v)
        x *= 2;  // altera só a cópia (o -Wall avisa: "set but not used")
}

void mostrar(const char* rotulo, const std::vector<int>& v) {
    std::print("{}", rotulo);
    for (int x : v)
        std::print(" {}", x);
    std::println("");
}

int main() {
    int a = 1, b = 2;
    trocar(a, b);  // em C seria trocar(&a, &b)
    std::println("a={} b={}", a, b);

    std::vector<int> v{1, 2, 3};
    dobrar_errado(v);
    mostrar("errado:", v);
    dobrar_todos(v);
    mostrar("certo: ", v);
}
```

Saída

```
a=2 b=1
errado: 1 2 3
certo:  2 4 6
```

A chamada `trocar(a, b)` não mostra que `a` e `b` serão alterados: esse é o preço da referência. Por isso há quem prefira ponteiro para parâmetros de saída, deixando o `&` visível na chamada. A diretriz atual (Core

Guidelines F.17) é usar referência não constante para "entra e sai" e devolver valores para "só sai". O compilador ainda avisa no `dobrar_errado` que `x` é alterado e nunca lido: vale ler os avisos.

O C++23 também sabe formatar contêineres direto (`std::println("{}", v)` imprime `[1, 2, 3]`), mas a libstdc++ só implementou isso na versão 15; com a 14, usada aqui, o laço do `mostrar` é necessário.

Q9 Sobrecarga para áreas

> *Escreva três funções `area` sobrecarregadas: uma para um círculo (raio `double`), uma para um retângulo (largura e altura `double`) e uma para um quadrado de lado inteiro (`int`). Mostre qual é chamada para `area(2)`, `area(2.0)` e `area(2.0, 3.0)`, e explique o que muda se a versão `int` for removida.*

```
#include <numbers>
#include <print>

double area(double raio) { return std::numbers::pi * raio * raio; }
double area(double l, double a) { return l * a; }
int area(int lado) { return lado * lado; }

int main() {
    std::println("area(2)        = {}", area(2));         // exata: int
    std::println("area(2.0)      = {:.4f}", area(2.0));  // exata: double
    std::println("area(2.0, 3.0) = {}", area(2.0, 3.0));
}
```

Saída

```
area(2)        = 4
area(2.0)      = 12.5664
area(2.0, 3.0) = 6
```

Sem a versão `int`, `area(2)` passaria a chamar a do círculo (conversão `int → double`) e imprimiria 12.5664: o programa continua compilando e muda de comportamento em silêncio. É o argumento contra sobrecargas com significados diferentes sob o mesmo nome; nomes como `area_circulo` evitariam a armadilha.

Q10 Contando palavras com std::string

> *Escreva `std::string normalizar(std::string_view s)` que remova espaços do começo e do fim e reduza espaços repetidos a um só. Depois conte as palavras da frase normalizada. Não use `char*` nem `strtok`.*

```cpp
#include <cctype>
#include <print>
#include <string>
#include <string_view>

std::string normalizar(std::string_view s) {
    std::string r;
    for (char c : s) {
        bool espaco = std::isspace(static_cast<unsigned char>(c));
        if (espaco) {
            if (!r.empty() && r.back() != ' ')
                r += ' ';
        } else {
            r += c;
        }
    }
    if (!r.empty() && r.back() == ' ')
        r.pop_back();
    return r;
}

int main() {
    std::string frase = "   C++   não é   C com classes  ";
    std::string n = normalizar(frase);
    int palavras = n.empty() ? 0 : 1;
    for (char c : n)
        if (c == ' ')
            ++palavras;
    std::println("[{}] tem {} palavras", n, palavras);
}
```

Saída

```
[C++ não é C com classes] tem 6 palavras
```

O `static_cast<unsigned char>` não é enfeite: as funções de `<cctype>` têm comportamento indefinido para `char` negativo, que é como os bytes de "ã" e "é" chegam em UTF-8 numa plataforma com `char` com sinal. A `std::string` cuida da memória: não há `malloc`, tamanho máximo nem `'\0'` para lembrar.

Q11 auto e o tipo verdadeiro

Escreva um programa que percorra um `std::map<std::string, int>` com `for (auto [nome, idade] : pessoas)` e mostre que alterar `idade` dentro do laço não muda o mapa. Corrija com a forma de referência e explique por que `nome` não pode ser alterado nem nela.

```cpp
#include <map>
#include <print>
#include <string>

int main() {
    std::map<std::string, int> pessoas{{"Ana", 30}, {"Bruno", 25}};

    for (auto [nome, idade] : pessoas)  // cópia de cada par
        ++idade;
    std::println("cópia:      Ana={} Bruno={}", pessoas["Ana"], pessoas["Bruno"]);

    for (auto& [nome, idade] : pessoas)  // referência ao par guardado no mapa
        ++idade;
    std::println("referência: Ana={} Bruno={}", pessoas["Ana"], pessoas["Bruno"]);
}
```

Saída

```
cópia:      Ana=30 Bruno=25
referência: Ana=31 Bruno=26
```

Cada elemento do mapa é um `std::pair<const std::string, int>`. A chave é `const` porque o mapa a mantém ordenada: alterá-la no lugar quebraria a ordem da árvore. Com `auto&`, `nome` vira referência para `const std::string`, e `nome += "x"` não compila. Para "mudar" uma chave, remove-se o elemento e insere-se outro (ou usa-se `extract`, do C++17).

MÓDULO 2

Objetos e posse de recursos

Construtores e destrutores, RAII, a regra dos três e dos cinco, ponteiros inteligentes, invariantes e inicialização de membros.

Artigos 07 a 12

#07 Nascimento e Morte de um Objeto — Construtores, Destrutores e o RAII na Prática
#08 Quando Copiar Dá Errado — a Regra dos Três e dos Cinco
#09 Um Dono, Um Recurso — o std::unique_ptr
#10 Posse Compartilhada e seus Perigos — shared_ptr e weak_ptr
#11 Projetando Abstrações — Encapsulamento, Invariantes e o Ponteiro this
#12 A Vida Começa Aqui — Inicialização de Membros a Fundo

Perguntas conceituais — respostas

Q12 RAII em uma frase

> *O que significa RAII, e por que ele substitui com vantagem o padrão do C de "`goto fim` e liberar tudo no final"?*

Resource Acquisition Is Initialization: **o recurso pertence a um objeto**; o construtor adquire e o destrutor libera. Como o destrutor de um objeto local roda **sempre** que o escopo termina (fim normal, `return` no meio, `break` ou exceção), a liberação não depende de o programador lembrar de cada caminho de saída. Em C, cada `return` antecipado é um vazamento em potencial, e o `goto fim` só funciona enquanto todo mundo segue a convenção. Em C++, quem escreve a classe acerta a liberação uma vez, e todos os usos ficam corretos.

Q13 Regra dos três, dos cinco e do zero

> *Uma classe guarda um `char*` alocado com `new[]`. Quais funções especiais ela precisa escrever (regra dos cinco)? E o que diz a regra do zero?*

Se a classe administra o recurso diretamente, as cópias geradas pelo compilador copiam **o ponteiro**, não o conteúdo: dois objetos liberariam a mesma memória (*double free*). Ela precisa definir:

1. o **destrutor** (`delete[]`);
2. o **construtor de cópia** (aloca e copia o conteúdo);
3. a **atribuição por cópia** (cuidado com `a = a` e com não vazar o buffer antigo);
4. o **construtor de movimento** (rouba o ponteiro e zera o da origem);
5. a **atribuição por movimento**.

Os três primeiros formam a regra dos três (anterior ao C++11); os dois de movimento completam a dos cinco. A **regra do zero** diz que o ideal é não escrever nenhum: guarde o recurso num membro que já sabe se cuidar (`std::string`, `std::vector`, `std::unique_ptr`) e o compilador gera os cinco corretamente.

Q14 unique_ptr não copia

Por que `std::unique_ptr` não pode ser copiado, mas pode ser movido? O que acontece com a origem depois de `auto b = std::move(a);`?

O `unique_ptr` expressa **posse exclusiva**: um só dono, responsável pelo `delete`. Uma cópia criaria dois donos e, portanto, dois `delete`. Por isso o construtor de cópia é `= delete`. Mover **transfere** a posse sem duplicá-la: depois de `auto b = std::move(a);`, `b` aponta para o objeto e `a` fica nulo (`a == nullptr`). O `std::move` em si não move nada: só converte `a` para uma referência *rvalue*, o que autoriza o construtor de movimento a "roubar" o ponteiro.

Q15 Ciclo de shared_ptr

Dois objetos `No` se apontam mutuamente com `std::shared_ptr`. Por que nenhum deles é destruído quando as variáveis locais saem de escopo? Como o `std::weak_ptr` resolve isso?

O `shared_ptr` destrói o objeto quando a **contagem de donos** chega a zero. No ciclo, quando as variáveis locais morrem, cada nó ainda é mantido vivo pelo outro: as contagens ficam em 1 para sempre, e a memória vaza (não há coletor de lixo para detectar ciclos). Um `weak_ptr` **observa sem ser dono**: não entra na contagem. Basta que um dos lados (tipicamente o "pai" visto do filho) seja `weak_ptr`; para usá-lo, chama-se `lock()`, que devolve um `shared_ptr` (nulo, se o objeto já morreu).

Q16 Ordem de inicialização dos membros

O que esta classe imprime para `Janela j(800);`, e por que o compilador avisa?

```
struct Janela {
    int area;
    int largura;
    Janela(int l) : largura(l), area(largura * 2) {}
};
```

Os membros são inicializados **na ordem em que foram declarados na classe**, não na ordem da lista de inicialização. `area` é declarado primeiro, então `area(largura * 2)` roda antes de `largura` receber 800 e lê um `int` não inicializado: comportamento indefinido, e o valor de `area` é lixo. O aviso (`-Wreorder`, ligado pelo `-Wall`) é justamente sobre a lista estar fora da ordem de declaração. A correção é declarar `largura` antes ou calcular `area(l * 2)` a partir do parâmetro.

Q17 explicit

Para que serve `explicit` num construtor de um argumento? Mostre o erro que ele previne com `struct Metros { Metros(double v); };` e uma função `void correr(Metros m)`.

Sem `explicit`, um construtor de um argumento vira uma **conversão implícita**: `correr(5.0)` compila e cria `Metros(5.0)` em silêncio, quando talvez o 5 estivesse em quilômetros ou segundos. Com `explicit Metros(double v);`, a chamada precisa ser `correr(Metros{5.0})`, e a unidade fica visível no código. A regra geral (Core Guidelines C.46) é marcar como `explicit` todo construtor de um argumento, salvo quando a conversão é realmente desejada, como de `const char*` para `std::string`.

Exercícios de código — respostas

Q18 Um arquivo com RAII

Escreva uma classe `Arquivo` que abra um `FILE*` no construtor (lançando `std::runtime_error` se falhar) e feche no destrutor, imprimindo uma mensagem em cada um. Proíba a cópia. Use-a numa função que retorna no meio e mostre que o arquivo é fechado mesmo assim.

```
#include <cstdio>
#include <print>
#include <stdexcept>
#include <string>

class Arquivo {
public:
    Arquivo(const std::string& nome, const char* modo) : f_(std::fopen(nome.c_str(), modo)),
nome_(nome) {
        if (!f_)
            throw std::runtime_error("não abriu " + nome);
        std::println("abriu {}", nome_);
    }
    ~Arquivo() {
        std::fclose(f_);
        std::println("fechou {}", nome_);
    }
    Arquivo(const Arquivo&) = delete;
    Arquivo& operator=(const Arquivo&) = delete;

    void escrever(const std::string& s) { std::fputs(s.c_str(), f_); }

private:
    std::FILE* f_;
    std::string nome_;
};

bool gravar(int n) {
    Arquivo a("saida.txt", "w");
    a.escrever("início\n");
    if (n < 0) {
        std::println("n negativo: retorno antecipado");
        return false;  // o destrutor de a roda aqui
    }
    a.escrever("fim\n");
    return true;
}

int main() {
    gravar(-1);
    try {
        Arquivo x("/pasta/que/nao/existe.txt", "r");
    } catch (const std::exception& e) {
        std::println("erro: {}", e.what());
    }
}
```

Saída

```
abriu saida.txt
n negativo: retorno antecipado
fechou saida.txt
erro: não abriu /pasta/que/nao/existe.txt
```

Quando o construtor lança, o objeto **nunca existiu**, e o destrutor não roda; por isso não aparece "fechou" para o arquivo inexistente, e o `fclose(nullptr)` nunca acontece. A biblioteca padrão já oferece `std::ofstream`, que segue o mesmo padrão; escrever o seu uma vez mostra o que ele faz por dentro.

Q19 Um buffer com a regra dos cinco

Escreva `class Buffer` que guarde `n` inteiros alocados com `new int[n]`, com os cinco membros especiais. Teste cópia, movimento e autoatribuição, e mostre o tamanho da origem depois de movida.

A atribuição usa o idioma **copiar e trocar**: recebe o parâmetro por valor (o que já faz a cópia ou o movimento) e troca os membros. Ele resolve autoatribuição e segurança contra exceções de uma vez.

```
#include <algorithm>
#include <cstddef>
#include <print>
#include <utility>

class Buffer {
public:
    explicit Buffer(std::size_t n) : n_(n), dados_(new int[n]{}) {}
    ~Buffer() { delete[] dados_; }

    Buffer(const Buffer& o) : n_(o.n_), dados_(new int[o.n_]) {
        std::copy(o.dados_, o.dados_ + n_, dados_);
    }
    Buffer(Buffer&& o) noexcept : n_(std::exchange(o.n_, 0)), dados_(std::exchange(o.dados_,
nullptr)) {}

    Buffer& operator=(Buffer o) noexcept {  // por valor: copia ou move conforme o argumento
        std::swap(n_, o.n_);
        std::swap(dados_, o.dados_);
        return *this;  // o antigo conteúdo morre com o
    }

    int& operator[](std::size_t i) { return dados_[i]; }
    std::size_t tamanho() const { return n_; }

private:
    std::size_t n_;
    int* dados_;
};

int main() {
    Buffer a(3);
    a[0] = 7;
    Buffer b = a;  // cópia
    b[0] = 9;
    std::println("a[0]={} b[0]={}", a[0], b[0]);

    Buffer c = std::move(a);  // movimento
    std::println("c[0]={} c.tamanho={} a.tamanho={}", c[0], c.tamanho(), a.tamanho());

    Buffer& mesmo = c;
    c = mesmo;  // autoatribuição, como em v[i] = v[j] com i == j
    std::println("depois de c = c: c[0]={}", c[0]);
    b = Buffer(5);
    std::println("b.tamanho={}", b.tamanho());
}
```

Saída

```
a[0]=7 b[0]=9
c[0]=7 c.tamanho=3 a.tamanho=0
depois de c = c: c[0]=7
b.tamanho=5
```

O `noexcept` no movimento importa: o `std::vector<Buffer>` só move os elementos ao crescer se o movimento não puder lançar; caso contrário, copia, para manter a garantia forte. Compile com `-fsanitize=address` e nenhum erro aparece. Na prática, um `std::vector<int>` como membro faria tudo isso sozinho (regra do zero).

Q20 Uma fábrica com unique_ptr

Escreva uma hierarquia mínima `Forma` (com destrutor virtual e `area()` virtual pura), `Circulo` e `Quadrado`, e uma função `std::unique_ptr<Forma> criar(char tipo, double medida)` que devolva `nullptr` para tipo desconhecido. Guarde as formas num `std::vector<std::unique_ptr<Forma>>` e mostre que todas são destruídas sem nenhum `delete` no código.

```cpp
#include <memory>
#include <numbers>
#include <print>
#include <vector>

struct Forma {
    virtual ~Forma() { std::println("~Forma"); }
    virtual double area() const = 0;
};

struct Circulo : Forma {
    double r;
    explicit Circulo(double r) : r(r) {}
    double area() const override { return std::numbers::pi * r * r; }
};

struct Quadrado : Forma {
    double l;
    explicit Quadrado(double l) : l(l) {}
    double area() const override { return l * l; }
};

std::unique_ptr<Forma> criar(char tipo, double medida) {
    switch (tipo) {
    case 'c': return std::make_unique<Circulo>(medida);
    case 'q': return std::make_unique<Quadrado>(medida);
    default: return nullptr;
    }
}

int main() {
    std::vector<std::unique_ptr<Forma>> formas;
    for (auto [t, m] : {std::pair{'c', 1.0}, {'q', 2.0}, {'x', 3.0}}) {
        if (auto f = criar(t, m))
            formas.push_back(std::move(f));  // sem o move, não compila: unique_ptr não copia
        else
            std::println("tipo desconhecido: {}", t);
    }
    for (const auto& f : formas)
        std::println("área = {:.2f}", f->area());
    std::println("fim do main");
}
```

Saída

```
tipo desconhecido: x
área = 3.14
área = 4.00
fim do main
~Forma
~Forma
```

Os dois `~Forma` aparecem **depois** de "fim do main": quando o vetor é destruído, cada `unique_ptr` destrói a sua forma. Sem o `virtual` no destrutor da base, apagar um `Circulo` por um ponteiro para `Forma` seria comportamento indefinido.

Q21 Quebrando o ciclo com weak_ptr

> *Monte uma árvore com `struct No { std::string nome; std::vector<std::shared_ptr<No>> filhos; std::weak_ptr<No> pai; }` e um destrutor que imprima o nome. Crie uma raiz com dois filhos, mostre o caminho de um filho até a raiz usando `pai.lock()` e confirme que tudo é destruído no fim. Depois diga o que aconteceria com `pai` sendo `shared_ptr`.*

```
#include <memory>
#include <print>
#include <string>
#include <vector>

struct No {
    std::string nome;
    std::vector<std::shared_ptr<No>> filhos;
    std::weak_ptr<No> pai;
    explicit No(std::string n) : nome(std::move(n)) {}
    ~No() { std::println("~No {}", nome); }
};

std::shared_ptr<No> adicionar(const std::shared_ptr<No>& pai, std::string nome) {
    auto f = std::make_shared<No>(std::move(nome));
    f->pai = pai;
    pai->filhos.push_back(f);
    return f;
}

int main() {
    auto raiz = std::make_shared<No>("raiz");
    adicionar(raiz, "a");
    auto b = adicionar(raiz, "b");

    std::print("caminho:");
    for (auto n = b; n; n = n->pai.lock())
        std::print(" {}", n->nome);
    std::println("");
    std::println("donos de b: {}", b.use_count());
}
```

Saída

```
caminho: b raiz
donos de b: 2
~No raiz
~No a
~No b
```

`b` tem dois donos: a variável local e o vetor `filhos` da raiz. O `weak_ptr` do filho para o pai não conta. Se `pai` fosse `shared_ptr`, a raiz teria como donos a variável `raiz` e os dois filhos; ao fim do `main`, a contagem dela cairia para 2, nunca para 0, e **nenhum** destrutor seria chamado. A saída não teria nenhuma linha `~No`, e o AddressSanitizer acusaria o vazamento.

Q22 Invariante protegida

Escreva `class Fracao` que mantenha sempre a invariante "denominador positivo e fração irredutível". O construtor deve lançar `std::invalid_argument` para denominador zero. Ofereça `numerador()`, `denominador()` e `somar(const Fracao&)`. Mostre `Fracao(2, -4)` e a soma de 1/6 com 1/3.

```
#include <numeric>
#include <print>
#include <stdexcept>

class Fracao {
public:
    Fracao(long n, long d) : n_(n), d_(d) {
        if (d_ == 0)
            throw std::invalid_argument("denominador zero");
        normalizar();
    }
    long numerador() const { return n_; }
    long denominador() const { return d_; }
    Fracao somar(const Fracao& o) const { return Fracao(n_ * o.d_ + o.n_ * d_, d_ * o.d_); }

private:
    void normalizar() {
        if (d_ < 0) {
            n_ = -n_;
            d_ = -d_;
        }
        long g = std::gcd(n_, d_);  // gcd(0, d) == d, então 0/5 vira 0/1
        n_ /= g;
        d_ /= g;
    }
    long n_, d_;
};

int main() {
    Fracao f(2, -4);
    std::println("2/-4 vira {}/{}", f.numerador(), f.denominador());
    Fracao s = Fracao(1, 6).somar(Fracao(1, 3));
    std::println("1/6 + 1/3 = {}/{}", s.numerador(), s.denominador());
    try {
        Fracao z(1, 0);
    } catch (const std::invalid_argument& e) {
        std::println("erro: {}", e.what());
    }
}
```

Saída

```
2/-4 vira -1/2
1/6 + 1/3 = 1/2
erro: denominador zero
```

Como os membros são privados e todo caminho de criação passa pelo construtor, **nenhum** objeto `Fracao` pode existir fora da invariante. É isso que justifica a classe: com uma `struct` de membros públicos, qualquer linha do programa poderia escrever `d = 0`.

MÓDULO 3

Herança, polimorfismo e operadores

A cadeia de construção, funções virtuais, classes abstratas, destrutores virtuais e a sobrecarga de operadores.

Artigos 13 a 16

#13 Uma Classe que Nasce de Outra — Herança e a Cadeia de Construção
#14 Um Método, Vários Comportamentos — Funções Virtuais e Polimorfismo
#15 Contratos e Destruição Segura — Classes Abstratas e Destrutores Virtuais
#16 Ensinando Símbolos ao seu Tipo — Sobrecarga de Operadores

Perguntas conceituais — respostas

Q23 Ordem de construção e destruição

> *Numa hierarquia `Base → Meio → Folha`, em que ordem rodam os construtores e os destrutores ao criar e destruir uma `Folha`? Por que essa ordem é a única segura?*

Construtores: **Base, Meio, Folha**. Destrutores: **Folha, Meio, Base**, a ordem inversa. Cada parte é construída sobre uma base que já está pronta: o construtor de `Meio` pode usar membros de `Base` com segurança. Na destruição, a parte mais derivada sai primeiro porque pode depender da base, que ainda está viva enquanto o destrutor dela roda. Os membros seguem a mesma regra: construídos na ordem de declaração, destruídos na inversa.

Q24 Fatiamento

> *O que é object slicing? Mostre como ele acontece com `void mostrar(Animal a)` recebendo um `Cachorro`, e como evitá-lo.*

Ao passar ou copiar um objeto derivado **por valor** para uma variável do tipo base, só a parte `Animal` é copiada: os membros de `Cachorro` são "fatiados" fora, e o objeto resultante é um `Animal` legítimo. Por isso, dentro de `mostrar`, a chamada virtual `a.som()` executa a versão de `Animal`, não a de `Cachorro`. O polimorfismo só funciona por **referência ou ponteiro**: `void mostrar(const Animal& a)`. Uma defesa adicional é tornar a base abstrata, ou apagar a cópia dela, para que o fatiamento nem compile.

Q25 Destrutor virtual

> *Quando o destrutor da base precisa ser `virtual`? O que acontece em `Base* p = new Derivada; delete p;` quando ele não é?*

Sempre que objetos derivados puderem ser destruídos **por um ponteiro para a base** (e isso inclui `std::unique_ptr<Base>`). Sem `virtual`, o `delete p` chama só `~Base`: é **comportamento indefinido** pelo padrão, e na prática o destrutor de `Derivada` não roda e os recursos dela vazam. A regra das Core Guidelines (C.35) é: o destrutor de uma classe base deve ser público e virtual, ou protegido e não virtual (quando a base não deve ser usada para destruir).

Q26 override e final

O que `override` e `final` acrescentam, se o programa compila sem eles? Dê um erro que o `override` pega.

Os dois transformam intenção em verificação. Com `override`, o compilador **confere** que a função realmente sobrescreve uma virtual da base. Sem ele, um detalhe na assinatura cria uma função nova em silêncio: se a base declara `virtual void desenhar() const` e a derivada escreve `void desenhar()` (sem o `const`), a derivada ganha uma segunda função, e chamadas pela base continuam indo para a versão antiga. Com `override`, isso é erro de compilação. O `final` impede que uma classe seja herdada ou que uma virtual seja sobrescrita de novo, e ainda permite ao compilador trocar a chamada virtual por uma direta (*devirtualização*).

Q27 Operadores membros ou livres

Por que `operator+` de um tipo `Vetor2` costuma ser uma função livre, enquanto `operator+=` é membro? E por que `operator<<` para `std::ostream` não pode ser membro de `Vetor2`?

O `+=` altera o objeto da esquerda, então é natural como membro, com acesso ao estado e devolvendo `*this` por referência. O `+` cria um valor novo e deve tratar os dois operandos de forma **simétrica**: como função livre, conversões implícitas valem para os dois lados, e ele pode ser escrito em termos do `+=` (`Vetor2 r = a; r += b; return r;`). No `operator<<`, o operando da esquerda é o `std::ostream`: um operador membro sempre tem o próprio objeto como operando da esquerda, então ele teria de ser membro de `std::ostream`, classe que você não pode alterar. Por isso é uma função livre (às vezes `friend`).

Exercícios de código — respostas

Q28 A cadeia de construção

Escreva `Base`, `Meio` e `Folha`, cada uma com um membro do tipo `Rastro` (uma classe que imprime ao nascer e ao morrer) e construtores e destrutores que imprimem. Crie uma `Folha` e explique a saída.

```
#include <print>
#include <string>

struct Rastro {
    std::string n;
    explicit Rastro(std::string nome) : n(std::move(nome)) { std::println("  membro {}", n); }
    ~Rastro() { std::println("  ~membro {}", n); }
};

struct Base {
    Rastro r{"de Base"};
    Base() { std::println("Base()"); }
    virtual ~Base() { std::println("~Base()"); }
};

struct Meio : Base {
    Rastro r{"de Meio"};
    Meio() { std::println("Meio()"); }
    ~Meio() override { std::println("~Meio()"); }
};

struct Folha : Meio {
    Folha() { std::println("Folha()"); }
    ~Folha() override { std::println("~Folha()"); }
};

int main() {
    Folha f;
    std::println("--- fim do escopo");
}
```

Saída

```
  membro de Base
Base()
  membro de Meio
Meio()
Folha()
--- fim do escopo
~Folha()
~Meio()
  ~membro de Meio
~Base()
  ~membro de Base
```

Para cada classe, os membros são construídos **antes** do corpo do construtor dela e destruídos **depois** do corpo do destrutor. Por isso "membro de Meio" aparece depois de `Base()` e antes de `Meio()`. O `Meio::r` esconde o `Base::r`, mas os dois existem: um `Meio` tem dois `Rastro`.

Q29 Polimorfismo e fatiamento

Escreva `Animal` com `virtual std::string som() const`, as derivadas `Cachorro` e `Gato`, e duas funções: `falar_valor(Animal a)` e `falar_ref(const Animal& a)`. Chame as duas com um `Cachorro` e mostre a diferença. Depois percorra um `std::vector<std::unique_ptr<Animal>>`.

```
#include <memory>
#include <print>
#include <string>
#include <vector>

struct Animal {
    virtual ~Animal() = default;
    virtual std::string som() const { return "..."; }
};

struct Cachorro : Animal {
    std::string som() const override { return "au"; }
};

struct Gato : Animal {
    std::string som() const override { return "miau"; }
};

void falar_valor(Animal a) { std::println("por valor:      {}", a.som()); }
void falar_ref(const Animal& a) { std::println("por referência: {}", a.som()); }

int main() {
    Cachorro c;
    falar_valor(c);  // fatiado: vira um Animal
    falar_ref(c);

    std::vector<std::unique_ptr<Animal>> bichos;
    bichos.push_back(std::make_unique<Cachorro>());
    bichos.push_back(std::make_unique<Gato>());
    for (const auto& b : bichos)
        std::println("{}", b->som());
}
```

Saída

```
por valor:      ...
por referência: au
au
miau
```

A chamada virtual escolhe a função pelo **tipo dinâmico** do objeto, guardado num ponteiro escondido para a tabela de funções virtuais (*vptr*). No `falar_valor`, o parâmetro é um `Animal` novo, construído por cópia, e o *vptr* dele aponta para a tabela de `Animal`.

Q30 Uma interface abstrata

Defina a interface `Notificador` com `virtual void enviar(const std::string& msg) = 0` e duas implementações, `Email` e `Sms` (que recusa mensagens com mais de 20 caracteres, lançando `std::length_error`). Escreva `void alertar(Notificador& n, const std::string& msg)` que trate o erro, e mostre que `Notificador n;` não compila.

```
#include <print>
#include <stdexcept>
#include <string>

class Notificador {
public:
    virtual ~Notificador() = default;
    virtual void enviar(const std::string& msg) = 0;
};

class Email : public Notificador {
public:
    void enviar(const std::string& msg) override { std::println("[email] {}", msg); }
};

class Sms : public Notificador {
public:
    void enviar(const std::string& msg) override {
        if (msg.size() > 20)
            throw std::length_error("SMS longo demais");
        std::println("[sms] {}", msg);
    }
};

void alertar(Notificador& n, const std::string& msg) {
    try {
        n.enviar(msg);
    } catch (const std::length_error& e) {
        std::println("falhou: {}", e.what());
    }
}

int main() {
    Email e;
    Sms s;
    alertar(e, "Servidor fora do ar desde 03:12");
    alertar(s, "Servidor caiu");
    alertar(s, "Servidor fora do ar desde 03:12");
    // Notificador n;  // erro: classe abstrata (enviar é virtual pura)
}
```

Saída

```
[email] Servidor fora do ar desde 03:12
[sms] Servidor caiu
falhou: SMS longo demais
```

A função `alertar` depende só do contrato, não das implementações: um `Push` novo entra sem mudar uma linha dela. Note que `msg.size()` conta **bytes**; uma mensagem com acentos chega ao limite com menos de 20 letras.

Q31 Operadores de um vetor 2D

Escreva `struct Vetor2 { double x, y; }` com `+=`, `+`, `-`, multiplicação por escalar dos dois lados (`v * 2` e `2 * v`), `==` (use `= default`) e um `std::formatter` para que `std::println("{}", v)` imprima `(x, y)`.

```cpp
#include <format>
#include <print>

struct Vetor2 {
    double x{}, y{};
    Vetor2& operator+=(const Vetor2& o) {
        x += o.x;
        y += o.y;
        return *this;
    }
    bool operator==(const Vetor2&) const = default;
};

Vetor2 operator+(Vetor2 a, const Vetor2& b) { return a += b; }
Vetor2 operator-(const Vetor2& a, const Vetor2& b) { return {a.x - b.x, a.y - b.y}; }
Vetor2 operator*(const Vetor2& v, double k) { return {v.x * k, v.y * k}; }
Vetor2 operator*(double k, const Vetor2& v) { return v * k; }

template <>
struct std::formatter<Vetor2> : std::formatter<double> {
    auto format(const Vetor2& v, std::format_context& ctx) const {
        auto out = std::format_to(ctx.out(), "(");
        ctx.advance_to(out);
        out = std::formatter<double>::format(v.x, ctx);  // respeita {:.2f} e afins
        out = std::format_to(out, ", ");
        ctx.advance_to(out);
        out = std::formatter<double>::format(v.y, ctx);
        return std::format_to(out, ")");
    }
};

int main() {
    Vetor2 a{1, 2}, b{0.5, -1};
    std::println("a + b   = {}", a + b);
    std::println("a - b   = {}", a - b);
    std::println("2 * a   = {}", 2 * a);
    std::println("a * 1/3 = {:.3f}", a * (1.0 / 3));
    std::println("a == a*1? {}", a == a * 1);
}
```

Saída

```
a + b   = (1.5, 1)
a - b   = (0.5, 3)
2 * a   = (2, 4)
a * 1/3 = (0.333, 0.667)
a == a*1? true
```

Herdar de `std::formatter<double>` reaproveita o `parse` dele: a precisão de `{:.3f}` vale para cada coordenada sem nenhum código extra. O `operator+` recebe `a` por valor de propósito: a cópia já é o resultado, e o `+=` só a altera.

Q32 Comparação com a nave espacial

Escreva `struct Versao { int maior, menor, patch; }` com `auto operator<=>(const Versao&) const = default;` e ordene um vetor de versões com `std::sort`. Explique por que a comparação gerada é a certa aqui, e quando não seria.

```
#include <algorithm>
#include <print>
#include <vector>

struct Versao {
    int maior, menor, patch;
    auto operator<=>(const Versao&) const = default;  // gera <, <=, >, >=, == e !=
};

int main() {
    std::vector<Versao> vs{{1, 10, 0}, {1, 2, 3}, {2, 0, 0}, {1, 2, 10}};
    std::sort(vs.begin(), vs.end());
    for (const auto& v : vs)
        std::print("{}.{}.{} ", v.maior, v.menor, v.patch);
    std::println("");
    std::println("1.2.10 > 1.2.3? {}", Versao{1, 2, 10} > Versao{1, 2, 3});
}
```

Saída

```
1.2.3 1.2.10 1.10.0 2.0.0
1.2.10 > 1.2.3? true
```

A comparação padrão é **lexicográfica, na ordem de declaração dos membros**: compara `maior`; se empatar, `menor`; depois `patch`. É exatamente a ordem do SemVer, e ainda corrige o erro de comparar versões como texto (`"1.10.0" < "1.2.3"` é verdadeiro como string). Não serviria se a ordem dos membros não fosse a ordem de importância, ou se algum membro não devesse participar (um cache, um contador de acessos); aí se escreve o `<=>` à mão.

MÓDULO 4

A biblioteca padrão: contêineres, iteradores e algoritmos

vector a fundo, string e string_view, map, set e as versões hash, iteradores, <algorithm>, o for de intervalo e lambdas.

Artigos 17 a 23

#17 O Array que Cresce Sozinho — std::vector a Fundo
#18 Texto como Coleção — std::string Revisitada e o std::string_view
#19 Buscas Instantâneas — map, set e suas Versões Hash
#20 O Fio que Une Tudo — Iteradores como Conceito Central
#21 Pare de Escrever Laços — a Biblioteca <algorithm>
#22 Percorrendo com Elegância — o Range-Based for a Fundo
#23 Funções Anônimas e Capturas — Lambdas de Verdade

Perguntas conceituais — respostas

Q33 Capacidade do vector

Qual a diferença entre `size()` e `capacity()` num `std::vector`? Por que um `push_back` pode invalidar todos os ponteiros e iteradores para os elementos, e como o `reserve` ajuda?

`size()` é quantos elementos existem; `capacity()` é para quantos há memória reservada. Quando um `push_back` encontra `size() == capacity()`, o vetor aloca um bloco maior (tipicamente 1,5 ou 2 vezes), move os elementos para lá e libera o antigo: todo ponteiro, referência ou iterador para o bloco velho passa a apontar para memória liberada. O `reserve(n)` aloca de uma vez para `n` elementos; enquanto `size()` não passar de `n`, nenhuma realocação acontece. Isso evita cópias repetidas e mantém os iteradores válidos, desde que o limite seja conhecido.

Q34 O perigo do string_view

Por que esta função tem um bug, e por que ele não é detectado pelo compilador?

```
std::string_view saudacao(const std::string& nome) {
    std::string s = "Olá, " + nome;
    return s;
}
```

O `std::string_view` é só **um ponteiro e um tamanho**: não é dono dos caracteres. Aqui ele aponta para o buffer de `s`, uma variável local destruída no `return`. Quem chamar recebe uma *view* para memória liberada (*dangling*), e lê-la é comportamento indefinido; às vezes "funciona", o que torna o bug traiçoeiro. O compilador aceita porque `std::string` converte implicitamente para `string_view`. Alguns compiladores avisam (o clang tem `-Wdangling-gsl`), e o AddressSanitizer detecta o acesso em execução. Regra prática: `string_view` é ótimo como **parâmetro**, perigoso como retorno ou membro.

Q35 map ou unordered_map

Compare `std::map` e `std::unordered_map` em complexidade, ordem de iteração e requisitos sobre a chave. Quando o `map` é a melhor escolha mesmo sendo O(log n)?

- ** `map` **: árvore balanceada (rubro-negra). Busca, inserção e remoção em O(log n),** sempre**. Itera em ordem crescente de chave. A chave precisa de `operator<` (ou de um comparador).
- ** `unordered_map` *: tabela hash. O(1) em média, O(n) no pior caso (muitas colisões). Ordem de iteração indefinida, e que pode mudar após um *rehash*.* A chave precisa de `std::hash` e de `==`.

O `map` vence quando a **ordem importa** (listar em ordem, buscar faixas com `lower_bound` / `upper_bound`, achar o vizinho mais próximo), quando o pior caso precisa ser garantido, ou quando escrever um hash para a chave dá trabalho. Para coleções pequenas, um `vector` ordenado com busca binária costuma vencer os dois, pela localidade de cache.

Q36 Categorias de iteradores

Por que `std::sort` não funciona com `std::list`? O que isso diz sobre as categorias de iteradores?

O `std::sort` exige **iteradores de acesso aleatório**: precisa de `it + n` e de `it2 - it1` em tempo constante para particionar o intervalo. A `std::list` oferece só iteradores **bidirecionais** (`++` e `--`): chegar ao elemento n custa O(n). Os algoritmos declaram a categoria mínima de que precisam (*input*, *forward*, *bidirectional*, *random access* e, no C++20, *contiguous*), e cada contêiner fornece a sua. Por isso a lista tem um método próprio, `lista.sort()`, uma ordenação por fusão que só religa os nós.

Q37 Erase-remove

Por que `std::remove(v.begin(), v.end(), 0)` não diminui o tamanho do vetor? Como apagar de verdade, antes e depois do C++20?

Os algoritmos trabalham com **iteradores**, não com contêineres: não têm como mudar o tamanho de nada. O `std::remove` só **compacta** os elementos que ficam no começo e devolve um iterador para o novo fim lógico; o que vem depois tem valores não especificados. Antes do C++20, o idioma era `v.erase(std::remove(v.begin(), v.end(), 0), v.end());`. No C++20 existe `std::erase(v, 0)` (e `std::erase_if(v, pred)`), que faz as duas coisas e devolve quantos elementos saíram.

Q38 Captura por referência

Esta função devolve uma lambda. Por que chamá-la depois é comportamento indefinido, e como corrigir?

```
auto contador() {
    int n = 0;
    return [&n] { return ++n; };
}
```

A lambda captura `n` **por referência**, e `n` é uma variável local que morre quando `contador` retorna. A lambda devolvida guarda uma referência para memória que não existe mais. A correção é capturar por valor

e permitir que a cópia interna seja alterada: `return [n = 0]() mutable { return ++n; };`. Assim cada lambda carrega o próprio contador. Regra: captura por referência só quando a lambda não sobrevive ao escopo (como num `std::sort` ou `std::for_each` local).

Exercícios de código — respostas

Q39 Observando a capacidade

Escreva um programa que faça 17 `push_back` num `std::vector<int>` vazio, imprimindo o tamanho e a capacidade sempre que a capacidade mudar. Repita com `reserve(17)` antes. O que o resultado diz sobre a estratégia de crescimento da sua biblioteca?

```
#include <cstddef>
#include <print>
#include <vector>

void encher(bool reservar) {
    std::vector<int> v;
    if (reservar)
        v.reserve(17);
    std::size_t cap = v.capacity();
    std::print("{}: cap {}", reservar ? "com reserve" : "sem reserve", cap);
    for (int i = 0; i < 17; ++i) {
        v.push_back(i);
        if (v.capacity() != cap) {
            cap = v.capacity();
            std::print(" -> {}", cap);
        }
    }
    std::println("");
}

int main() {
    encher(false);
    encher(true);
}
```

Saída

```
sem reserve: cap 0 -> 1 -> 2 -> 4 -> 8 -> 16 -> 32
com reserve: cap 17
```

A libstdc++ (do gcc, usada também pelo clang no Linux) **dobra** a capacidade: seis realocações até 17 elementos, cada uma movendo tudo o que já existia. O MSVC cresce 1,5 vez, e o padrão só exige que o `push_back` seja O(1) amortizado, o que qualquer fator maior que 1 garante. Com `reserve`, nenhuma realocação.

Q40 Frequência de palavras

> *Leia um texto (pode estar numa `std::string` no programa), conte a frequência de cada palavra em minúsculas com um `std::unordered_map<std::string, int>` e mostre as três mais frequentes, desempatando em ordem alfabética. Use `std::istringstream` para separar as palavras.*

O `unordered_map` conta em O(1); para o ranking, os pares vão para um vetor, ordenado por um comparador que usa frequência decrescente e palavra crescente.

```
#include <algorithm>
#include <cctype>
#include <print>
#include <sstream>
#include <string>
#include <unordered_map>
#include <utility>
#include <vector>

int main() {
    std::string texto = "O rato roeu a roupa do rei. O rei riu, o rato fugiu e a roupa ficou.";
    std::unordered_map<std::string, int> freq;
    std::istringstream in(texto);
    std::string palavra;
    while (in >> palavra) {
        std::string limpa;
        for (unsigned char c : palavra)
            if (std::isalpha(c))
                limpa += static_cast<char>(std::tolower(c));
        if (!limpa.empty())
            ++freq[limpa];
    }

    std::vector<std::pair<std::string, int>> ranking(freq.begin(), freq.end());
    std::sort(ranking.begin(), ranking.end(), [](const auto& a, const auto& b) {
        return a.second != b.second ? a.second > b.second : a.first < b.first;
    });
    for (std::size_t i = 0; i < 3 && i < ranking.size(); ++i)
        std::println("{}: {}", ranking[i].first, ranking[i].second);
}
```

Saída

```
o: 3
a: 2
rato: 2
```

O desempate não é detalhe: sem ele, a ordem entre palavras com a mesma contagem dependeria da ordem interna da tabela hash, que muda entre bibliotecas e versões. O `isalpha` byte a byte funciona para este texto sem acentos; com "não" ou "ação", seria preciso tratar UTF-8 (ou converter para `std::u32string`).

Q41 Agenda ordenada com map

Use um `std::map<std::string, std::string>` (nome → telefone) para uma agenda. Insira cinco nomes fora de ordem, liste em ordem alfabética, busque um nome com `find` e liste só os nomes que começam com "Ma" usando `lower_bound`.

```
#include <map>
#include <print>
#include <string>

int main() {
    std::map<std::string, std::string> agenda{
        {"Marta", "9999-0001"}, {"Bruno", "9999-0002"}, {"Mateus", "9999-0003"},
        {"Ana", "9999-0004"}, {"Mário", "9999-0005"},
    };

    for (const auto& [nome, tel] : agenda)
        std::println("{:<7} {}", nome, tel);

    if (auto it = agenda.find("Bruno"); it != agenda.end())
        std::println("Bruno: {}", it->second);
    if (!agenda.contains("Zé"))
        std::println("Zé não está na agenda");

    std::print("começam com Ma:");
    for (auto it = agenda.lower_bound("Ma"); it != agenda.end() && it->first.starts_with("Ma");
++it)
        std::print(" {}", it->first);
    std::println("");
}
```

Saída

```
Ana     9999-0004
Bruno   9999-0002
Marta   9999-0001
Mateus  9999-0003
Mário   9999-0005
Bruno: 9999-0002
Zé não está na agenda
começam com Ma: Marta Mateus
```

O `lower_bound("Ma")` leva direto ao primeiro nome maior ou igual a "Ma" em O(log n), e o laço para no primeiro que não começa com o prefixo: não percorre a agenda inteira. O "Mário" ficou de fora: para o computador ele começa com "Má", outro prefixo. E a ordem é a dos **bytes**: "Mário" vem depois de "Mateus" porque o "á" em UTF-8 (0xC3 0xA1) é maior que qualquer letra ASCII. Para ordenar como num dicionário, é preciso uma comparação por *locale* ou uma biblioteca como a ICU.

Q42 Algoritmos no lugar de laços

Dado `std::vector<int> v{5, 3, 8, 1, 9, 2, 7}`, use só funções de `<algorithm>` e `<numeric>`, sem escrever nenhum `for`, para: (a) somar os elementos; (b) contar os pares; (c) achar o maior; (d) verificar se todos são positivos; (e) gerar um vetor com os quadrados; (f) ordenar de forma decrescente.

```
#include <algorithm>
#include <functional>
#include <iterator>
#include <numeric>
#include <print>
#include <vector>

int main() {
    std::vector<int> v{5, 3, 8, 1, 9, 2, 7};

    int soma = std::accumulate(v.begin(), v.end(), 0);
    auto pares = std::count_if(v.begin(), v.end(), [](int x) { return x % 2 == 0; });
    int maior = *std::max_element(v.begin(), v.end());
    bool positivos = std::all_of(v.begin(), v.end(), [](int x) { return x > 0; });

    std::vector<int> quad;
    std::transform(v.begin(), v.end(), std::back_inserter(quad), [](int x) { return x * x; });

    std::ranges::sort(v, std::greater{});

    std::println("soma={} pares={} maior={} positivos={}", soma, pares, maior, positivos);
    std::print("quadrados:");
    std::ranges::for_each(quad, [](int x) { std::print(" {}", x); });
    std::print("\ndecrescente:");
    std::ranges::for_each(v, [](int x) { std::print(" {}", x); });
    std::println("");
}
```

Saída

```
soma=35 pares=2 maior=9 positivos=true
quadrados: 25 9 64 1 81 4 49
decrescente: 9 8 7 5 3 2 1
```

Cada nome diz o que acontece (`count_if`, `all_of`), coisa que um `for` genérico só revela lendo o corpo. As versões de `std::ranges` (C++20) recebem o contêiner inteiro, o que elimina o erro de passar `begin` de um vetor e `end` de outro. O `std::back_inserter` faz o `push_back` por conta própria; sem ele, `transform` escreveria num vetor vazio, além do fim.

Q43 Removendo durante a iteração

Dado um `std::vector<std::string>` com nomes, apague todos os que têm menos de 4 letras de três formas: (a) o laço "ingênuo" com `erase` dentro de um `for` de intervalo (explique por que não fazer); (b) o laço correto com o iterador devolvido por `erase`; (c) `std::erase_if`.

O item (a) é um bug: o `erase` invalida o iterador que o `for` de intervalo usa por dentro, e o próximo `++` é comportamento indefinido. Por isso ele não aparece no programa.

```
#include <print>
#include <string>
#include <vector>

void mostrar(const std::vector<std::string>& v) {
    for (const auto& s : v)
        std::print("{} ", s);
    std::println("");
}

int main() {
    const std::vector<std::string> original{"Ana", "Bruno", "Lu", "Carla", "Edu", "Fernanda"};

    // (b) laço com o iterador devolvido por erase: O(n²) no pior caso
    auto v = original;
    for (auto it = v.begin(); it != v.end();) {
        if (it->size() < 4)
            it = v.erase(it);  // erase devolve o próximo elemento válido
        else
            ++it;
    }
    mostrar(v);

    // (c) C++20: uma passada só, O(n)
    auto w = original;
    auto removidos = std::erase_if(w, [](const std::string& s) { return s.size() < 4; });
    mostrar(w);
    std::println("removidos: {}", removidos);
}
```

Saída

```
Bruno Carla Fernanda
Bruno Carla Fernanda
removidos: 3
```

Além de correto, o `erase_if` é mais rápido: cada `erase` no meio de um vetor desloca todos os elementos seguintes, enquanto o `erase_if` desloca cada elemento uma vez só.

Q44 Lambdas que guardam estado

Escreva `auto criar_contador(int inicio, int passo)` que devolva uma lambda que, a cada chamada, devolve o próximo valor da sequência. Crie dois contadores independentes. Depois use uma lambda com captura por referência para somar, com `std::for_each`, os elementos de um vetor numa variável local.

```
#include <algorithm>
#include <print>
#include <vector>

auto criar_contador(int inicio, int passo) {
    return [atual = inicio, passo]() mutable {  // cópias próprias, que sobrevivem à função
        int r = atual;
        atual += passo;
        return r;
    };
}

int main() {
    auto de2 = criar_contador(0, 2);
    auto de10 = criar_contador(100, -10);
    for (int i = 0; i < 4; ++i)
        std::print("{} {} | ", de2(), de10());
    std::println("");

    std::vector<int> v{1, 2, 3, 4};
    int soma = 0;
    std::for_each(v.begin(), v.end(), [&soma](int x) { soma += x; });  // a lambda não sobrevive
a soma
    std::println("soma = {}", soma);
}
```

Saída

```
0 100 | 2 90 | 4 80 | 6 70 |
soma = 10
```

Sem o `mutable`, o `operator()` gerado para a lambda seria `const` e não poderia alterar `atual`. Cada lambda é um objeto de uma classe anônima com os membros capturados: `de2` e `de10` têm estados independentes, como dois objetos de uma mesma classe.

MÓDULO 5

Programação genérica

Templates de função e de classe, e concepts para dizer o que um tipo precisa oferecer.

Artigos 24 a 26

#24 Escreva uma Vez, Use com Todo Tipo — Templates de Função
#25 Estruturas de Dados para Qualquer Tipo — Templates de Classe
#26 Requisitos Explícitos — Concepts e o Fim dos Erros Ilegíveis

Perguntas conceituais — respostas

Q45 Template em vez de macro

Compare `#define MAX(a, b) ((a) > (b) ? (a) : (b))` com `template <typename T> const T& maximo(const T& a, const T& b)`. Cite dois problemas da macro que o template não tem.

- **Avaliação dupla**: `MAX(i++, j)` incrementa `i` duas vezes quando ele é o maior; o template avalia cada argumento uma vez, como qualquer função.
- **Sem tipos**: a macro compara um `int` com um `unsigned` sem queixa (e com o resultado errado para negativos); o template exige que `a` e `b` tenham o mesmo `T`, e `maximo(1, 2u)` é erro de compilação.

Além disso, o template respeita escopo e *namespaces*, aparece no depurador e dá erros sobre o código real, não sobre o texto já expandido.

Q46 Por que templates ficam no cabeçalho

Por que a definição de uma função template normalmente fica no `.h`, e não no `.cpp` como as funções comuns?

Um template não é código: é uma **receita** para gerar código. A função `maximo<int>` só é gerada (instanciada) quando o compilador vê um uso com `int` **e** tem a definição completa à mão. Como cada `.cpp` é compilado separadamente, a unidade que chama `maximo(3, 4)` precisa enxergar o corpo do template; se ele estiver só em `maximo.cpp`, nenhuma unidade gera `maximo<int>`, e o *linker* reclama de símbolo indefinido. A alternativa é a **instanciação explícita** (`template const int& maximo(const int&, const int&);` no `.cpp`), que só funciona quando os tipos são conhecidos de antemão.

Q47 O ganho dos concepts

O que muda para quem usa a função quando `template <typename T> T somar(...)` vira `template <std::integral T> T somar(...)`? Mostre a diferença na mensagem de erro de `somar(std::string("a"), ...)`.

Sem concept, o compilador aceita qualquer `T` na chamada e só descobre o problema **dentro do corpo**, ao instanciar: o erro aponta uma linha interna do template, às vezes a dezenas de níveis de profundidade, e cita tipos que o usuário nunca escreveu. Com `std::integral`, o requisito faz parte da **assinatura**: a chamada com `std::string` falha no próprio local da chamada, com uma mensagem do tipo "*constraints not satisfied:*

std::string does not satisfy integral". O concept também documenta a intenção e permite sobrecargas por requisito (uma versão para inteiros e outra para pontos flutuantes, por exemplo).

Exercícios de código — respostas

Q48 Um template de função com concept

Escreva `template <typename T> T limitar(T valor, T minimo, T maximo)` que prenda o valor ao intervalo, restrito a tipos com `operator<` (use `std::totally_ordered`). Teste com `int`, `double` e `std::string`, e mostre (em comentário) a chamada que não compila e por quê.

```
#include <concepts>
#include <print>
#include <string>

template <std::totally_ordered T>
T limitar(T valor, T minimo, T maximo) {
    if (valor < minimo)
        return minimo;
    if (maximo < valor)
        return maximo;
    return valor;
}

struct Ponto {
    int x, y;
};

int main() {
    std::println("{}", limitar(15, 0, 10));
    std::println("{}", limitar(-0.5, 0.0, 1.0));
    std::println("{}", limitar(std::string("m"), std::string("a"), std::string("f")));
    // limitar(5, 0.0, 10.0);           // erro: T deduzido como int e como double
    // limitar(Ponto{}, Ponto{}, Ponto{}); // erro: Ponto não satisfaz totally_ordered
}
```

Saída

```
10
0
f
```

O primeiro erro comentado não tem a ver com o concept: a dedução de `T` exige que os três argumentos concordem. `limitar<double>(5, 0.0, 10.0)` resolve, convertendo o 5. O mesmo algoritmo já existe na biblioteca padrão como `std::clamp`.

Q49 Uma pilha genérica

Escreva `template <typename T> class Pilha` sobre um `std::vector<T>`, com `empilhar`, `desempilhar` (que devolve `std::optional<T>`, vazio quando a pilha está vazia), `topo` e `tamanho`. Use-a com `int` e com `std::string`.

```cpp
#include <cstddef>
#include <optional>
#include <print>
#include <string>
#include <utility>
#include <vector>

template <typename T>
class Pilha {
public:
    void empilhar(T v) { dados_.push_back(std::move(v)); }

    std::optional<T> desempilhar() {
        if (dados_.empty())
            return std::nullopt;
        T v = std::move(dados_.back());
        dados_.pop_back();
        return v;
    }

    const T* topo() const { return dados_.empty() ? nullptr : &dados_.back(); }
    std::size_t tamanho() const { return dados_.size(); }

private:
    std::vector<T> dados_;
};

int main() {
    Pilha<int> p;
    for (int i = 1; i <= 3; ++i)
        p.empilhar(i * 10);
    while (auto v = p.desempilhar())
        std::print("{} ", *v);
    std::println("| vazia: {}", p.tamanho() == 0);

    Pilha<std::string> s;
    s.empilhar("primeiro");
    s.empilhar("segundo");
    if (const auto* t = s.topo())
        std::println("topo: {} (tamanho {})", *t, s.tamanho());
}
```

Saída

```
30 20 10 | vazia: true
topo: segundo (tamanho 2)
```

`empilhar` recebe por valor e move: quem passa um temporário paga um movimento; quem passa uma variável paga uma cópia, que de qualquer forma seria necessária. O `topo` devolve ponteiro porque "não há topo" é um caso legítimo, e um `std::optional<T&>` só chega no C++26.

Q50 Um concept próprio

Defina o concept `Desenhavel`: um tipo que tem `desenhar()` devolvendo algo conversível para `std::string` e `area()` devolvendo um tipo de ponto flutuante. Escreva `void relatorio(const Desenhavel auto& f)` e teste com dois tipos sem herança. Mostre por que um terceiro tipo é rejeitado.

```
#include <concepts>
#include <print>
#include <string>

template <typename T>
concept Desenhavel = requires(const T& t) {
    { t.desenhar() } -> std::convertible_to<std::string>;
    { t.area() } -> std::floating_point;
};

struct Quadrado {
    double lado;
    std::string desenhar() const { return "[]"; }
    double area() const { return lado * lado; }
};

struct Triangulo {
    double base, altura;
    const char* desenhar() const { return "/\\"; }
    float area() const { return static_cast<float>(base * altura / 2); }
};

struct Texto {
    std::string desenhar() const { return "abc"; }
    int area() const { return 0; }  // int não é floating_point
};

void relatorio(const Desenhavel auto& f) {
    std::println("{} área {:.1f}", std::string(f.desenhar()), f.area());
}

int main() {
    relatorio(Quadrado{3});
    relatorio(Triangulo{4, 5});
    static_assert(!Desenhavel<Texto>);
    // relatorio(Texto{});  // erro: Texto não satisfaz Desenhavel (area devolve int)
}
```

Saída

```
[] área 9.0
/\ área 10.0
```

É polimorfismo **em tempo de compilação**: nenhuma classe base, nenhuma tabela virtual, e cada chamada de `relatorio` gera uma função própria para o tipo. O preço é o tamanho do executável e a impossibilidade

de guardar quadrados e triângulos no mesmo `std::vector`, que é onde a herança do módulo 3 continua necessária.

MÓDULO 6

Erros, valores opcionais e o C++ moderno

Exceções e RAII, std::optional, std::variant, std::expected e structured bindings.

Artigos 27 a 30

#27 Quando o Contrato se Quebra — Exceções e a Aliança com o RAII
#28 Um Valor que Pode Não Existir — std::optional
#29 Um de Vários, ou o Valor e a Razão da Falha — std::variant e std::expected
#30 O Conforto do C++ Moderno — Structured Bindings e Dedução de Tipos

Perguntas conceituais — respostas

Q51 Exceção ou código de erro

> *Em C, erros voltam como valor de retorno. Que dois problemas desse modelo as exceções resolvem, e qual é o custo delas?*

- **Erro ignorado**: um código de retorno pode ser descartado em silêncio (quem confere o retorno do `printf`?). Uma exceção não tratada encerra o programa: não passa despercebida.
- **Propagação manual**: com códigos, cada nível intermediário precisa testar e repassar o erro, misturando o caminho de erro com a lógica. A exceção atravessa os níveis sozinha até quem sabe tratá-la, e o RAII limpa tudo no caminho.

O custo: lançar é **caro** (ordens de grandeza mais lento que um `return`), o fluxo de controle fica invisível na leitura, e o código precisa ser seguro contra exceções em todo lugar. Por isso exceção é para o **excepcional**; uma falha esperada, como "usuário digitou texto", combina mais com `std::optional` ou `std::expected`.

Q52 Garantias de segurança

> *Explique as garantias básica, forte e nothrow. Qual delas o `std::vector::push_back` oferece?*

- **Nothrow**: a operação nunca lança (destrutores, `swap`, movimentos marcados `noexcept`).
- **Forte**: se lançar, o estado volta a ser **exatamente** o de antes, como uma transação.
- **Básica**: se lançar, nada vaza e os objetos continuam válidos (invariantes preservadas), mas o estado pode ter mudado.

O `push_back` dá a garantia **forte**: se a cópia do novo elemento ou a realocação lançar, o vetor fica como estava. É por isso que, ao realocar, ele só move os elementos se o movimento for `noexcept`; senão, copia, para poder desistir sem estragar o original.

Q53 optional, variant ou expected

> *Para cada situação, escolha entre `std::optional<T>`, `std::variant<A, B>` e `std::expected<T, E>`, justificando: (a) buscar um usuário por id; (b) converter texto em número, informando por que falhou; (c) um token de uma calculadora, que pode ser número, operador ou parêntese.*

- (a) **`optional<Usuario>`**: "não encontrado" é uma resposta normal e não precisa de explicação.

- (b) **`expected<double, ErroConversao>` : ou o valor, ou o** motivo** da falha (vazio, caractere inválido, fora do intervalo). O `optional` perderia o motivo.
- (c) **`variant<double, char, Parentese>` : o token é sempre** um de vários tipos** igualmente legítimos; nenhum deles é "erro". Tratado com `std::visit` ou com `std::holds_alternative`.

O `expected` é, por dentro, um `variant` de dois tipos com papéis definidos: um esperado e um de erro.

Q54 **Structured bindings**

> *O que `auto [it, inseriu] = mapa.insert({"a", 1});` declara? Por que essa forma é melhor que `std::pair` com `.first` e `.second`?*

O `insert` devolve um `std::pair<iterator, bool>`: o iterador para o elemento (novo ou o que já existia) e se a inserção aconteceu. A declaração cria duas variáveis com **nomes que dizem o que são**. Com o par, o código fica cheio de `res.first->second` e `res.second`, que exigem lembrar o significado de cada posição. Funciona com pares, tuplas, arrays e *structs* de membros públicos.

Exercícios de código — respostas

Q55 **Exceções e o desempilhar**

> *Escreva três funções aninhadas (`nivel1` chama `nivel2`, que chama `nivel3`), cada uma com um objeto local que imprime ao ser destruído. O `nivel3` lança `std::runtime_error`; só o `main` trata. Mostre a ordem das mensagens e explique o stack unwinding.*

```cpp
#include <print>
#include <stdexcept>
#include <string>

struct Guarda {
    std::string n;
    ~Guarda() { std::println("  libera {}", n); }
};

void nivel3() {
    Guarda g{"nivel3"};
    std::println("nivel3 vai lançar");
    throw std::runtime_error("disco cheio");
}

void nivel2() {
    Guarda g{"nivel2"};
    nivel3();
    std::println("nunca impresso");
}

void nivel1() {
    Guarda g{"nivel1"};
    nivel2();
}

int main() {
    try {
        nivel1();
    } catch (const std::exception& e) {
        std::println("main tratou: {}", e.what());
    }
}
```

Saída

```
nivel3 vai lançar
  libera nivel3
  libera nivel2
  libera nivel1
main tratou: disco cheio
```

Entre o `throw` e o `catch`, o C++ **desempilha** as chamadas: cada função interrompida tem os objetos locais destruídos, do mais interno ao mais externo. É o RAII que torna isso seguro: arquivos, travas e memória dos três níveis são liberados sem um único `catch` intermediário. Se um destrutor lançasse durante esse processo, o programa chamaria `std::terminate`; por isso destrutores são `noexcept` por padrão.

Q56 Convertendo texto com expected

> *Escreva `std::expected<int, std::string> para_int(std::string_view s)` usando `std::from_chars`, com mensagens diferentes para texto vazio, caractere inválido e número fora do intervalo do `int`. Teste com `"42"`, `""`, `"12a"` e `"99999999999"`, e use `value_or` num dos casos.*

```cpp
#include <charconv>
#include <expected>
#include <print>
#include <string>
#include <string_view>
#include <system_error>

std::expected<int, std::string> para_int(std::string_view s) {
    if (s.empty())
        return std::unexpected("texto vazio");
    int v{};
    auto [fim, ec] = std::from_chars(s.data(), s.data() + s.size(), v);
    if (ec == std::errc::result_out_of_range)
        return std::unexpected("fora do intervalo de int");
    if (ec != std::errc{} || fim != s.data() + s.size())
        return std::unexpected("caractere inválido em \"" + std::string(s) + "\"");
    return v;
}

int main() {
    for (std::string_view s : {"42", "", "12a", "99999999999"}) {
        if (auto r = para_int(s))
            std::println("{:>13} -> {}", s, *r);
        else
            std::println("{:>13} -> erro: {}", s, r.error());
    }
    std::println("com padrão: {}", para_int("abc").value_or(-1));
}
```

Saída

```
           42 -> 42
              -> erro: texto vazio
          12a -> erro: caractere inválido em "12a"
  99999999999 -> erro: fora do intervalo de int
com padrão: -1
```

O `from_chars` não usa *locale*, não aloca e não lança: devolve o ponteiro onde parou e um código de erro. Conferir `fim` é essencial, porque ele aceita "12" em "12a" e para no "a". Compare com o `atoi` do C, que devolve 0 tanto para "0" quanto para "abc".

Q57 Encadeando com and_then

> *Usando o `para_int` do exercício anterior, escreva `std::expected<int, std::string> dividir(int a, int b)` (erro para `b == 0`) e calcule, com `and_then` e `transform`, a expressão "converter dois textos, dividir e dobrar o resultado", sem nenhum `if`. Teste um caminho de sucesso e dois de erro.*

```cpp
#include <charconv>
#include <expected>
#include <print>
#include <string>
#include <string_view>

std::expected<int, std::string> para_int(std::string_view s) {
    int v{};
    auto [fim, ec] = std::from_chars(s.data(), s.data() + s.size(), v);
    if (s.empty() || ec != std::errc{} || fim != s.data() + s.size())
        return std::unexpected("\"" + std::string(s) + "\" não é inteiro");
    return v;
}

std::expected<int, std::string> dividir(int a, int b) {
    if (b == 0)
        return std::unexpected("divisão por zero");
    return a / b;
}

std::expected<int, std::string> calcular(std::string_view x, std::string_view y) {
    return para_int(x).and_then([&](int a) {
        return para_int(y).and_then([a](int b) { return dividir(a, b); });
    }).transform([](int q) { return q * 2; });
}

int main() {
    for (auto [x, y] : {std::pair{"84", "4"}, {"84", "0"}, {"oito", "2"}}) {
        auto r = calcular(x, y);
        std::println("{} / {} * 2 -> {}", x, y, r ? std::to_string(*r) : "erro: " + r.error());
    }
}
```

Saída

```
84 / 4 * 2 -> 42
84 / 0 * 2 -> erro: divisão por zero
oito / 2 * 2 -> erro: "oito" não é inteiro
```

O `and_then` chama a função só se houver valor, e ela própria devolve um `expected` (pode falhar). O `transform` aplica uma função que não falha. No primeiro erro, os passos seguintes são pulados e o erro chega intacto ao fim: o mesmo efeito do `try/catch`, mas visível no tipo de retorno e sem o custo de uma exceção.

Q58 Tokens com variant

> *Escreva um tokenizador para expressões como `"3.5 * (2 + 10)"` que devolva um `std::vector<std::variant<double, char>>`, com números em `double` e operadores e parênteses em `char`. Imprima os tokens com `std::visit`, mostrando o tipo de cada um.*

```cpp
#include <cctype>
#include <print>
#include <string>
#include <string_view>
#include <type_traits>
#include <variant>
#include <vector>

using Token = std::variant<double, char>;

std::vector<Token> tokenizar(std::string_view s) {
    std::vector<Token> t;
    for (std::size_t i = 0; i < s.size();) {
        unsigned char c = s[i];
        if (std::isspace(c)) {
            ++i;
        } else if (std::isdigit(c) || c == '.') {
            std::size_t usados = 0;
            t.emplace_back(std::stod(std::string(s.substr(i)), &usados));
            i += usados;
        } else {
            t.emplace_back(static_cast<char>(c));
            ++i;
        }
    }
    return t;
}

int main() {
    for (const Token& tk : tokenizar("3.5 * (2 + 10)")) {
        std::visit([](auto v) {
            if constexpr (std::is_same_v<decltype(v), double>)
                std::print("[num {}] ", v);
            else
                std::print("[op {}] ", v);
        }, tk);
    }
    std::println("");
}
```

Saída

```
[num 3.5] [op *] [op (] [num 2] [op +] [num 10] [op )]
```

O `variant` sabe sempre qual tipo guarda, coisa que a `union` do C deixava a cargo do programador (com uma etiqueta separada que podia mentir). O `if constexpr` escolhe o ramo em tempo de compilação, para cada tipo que a lambda genérica recebe. O `stod` usa o *locale* do C e lança para texto inválido, como um "." sozinho; para entrada não confiável, o `from_chars` com `double` é a escolha mais segura.

MÓDULO 7

Concorrência

std::thread, mutex e lock_guard, std::atomic, std::async e condition_variable.

Artigos 31 a 34

#31 Fazendo Duas Coisas ao Mesmo Tempo — std::thread
#32 O Cadeado que Protege — std::mutex e lock_guard
#33 Operações Indivisíveis — std::atomic
#34 Tarefas em Segundo Plano e a Arte de Esperar — std::async e condition_variable

Perguntas conceituais — respostas

Q59 join ou detach

> *O que acontece se um `std::thread` é destruído sem `join()` nem `detach()`? Por que o `std::jthread` (C++20) existe?*

O destrutor de um `std::thread` ainda "juntável" chama ** `std::terminate` **, encerrando o programa: o padrão se recusa a escolher por você entre esperar (o que poderia travar) e soltar a thread (o que poderia deixá-la acessando variáveis já destruídas). O `std::jthread` faz o `join` no destrutor, aplicando RAII à thread, e ainda traz um `std::stop_token` para pedir que ela termine de forma cooperativa.

Q60 Condição de corrida

> *Duas threads executam `++contador` 100 mil vezes cada sobre um `int` compartilhado. Por que o resultado quase nunca é 200 mil? Cite duas correções.*

`++contador` não é uma operação só: é **ler, somar e escrever**. Se as duas threads leem o mesmo valor antes de qualquer uma escrever, um incremento se perde. Pelo padrão, acessar a mesma variável de duas threads, com ao menos uma escrita e sem sincronização, é uma **corrida de dados**: comportamento indefinido, não apenas um resultado impreciso. Correções:

- `std::atomic<int>`: o `++` vira uma instrução atômica de leitura, modificação e escrita;
- um `std::mutex` protegendo o incremento, com `std::lock_guard` ou `std::scoped_lock`.

Uma terceira, geralmente a mais rápida: cada thread soma num contador **local** e só no fim junta o total.

Q61 Deadlock

> *Duas threads precisam travar os mutexes `a` e `b`. A primeira trava `a` e depois `b`; a segunda, `b` e depois `a`. Explique o deadlock e duas formas de evitá-lo.*

Se a primeira trava `a` e a segunda trava `b` ao mesmo tempo, cada uma espera para sempre pelo mutex que a outra segura. Formas de evitar:

- **Ordem global**: toda thread trava os mutexes sempre na mesma ordem (por exemplo, por endereço).

- ** `std::scoped_lock lk(a, b);` * *(C++17): trava os dois com um algoritmo que evita o *deadlock*, qualquer que seja a ordem em que forem passados.

E, quando possível, o mais simples: não segurar dois mutexes ao mesmo tempo.

Q62 **Espera espúria**

> *Por que a espera numa `std::condition_variable` deve usar um predicado (`cv.wait(lk, [&]{ return pronto; })`) em vez de um `cv.wait(lk)` simples?*

Por dois motivos. **Despertar espúrio**: o padrão permite que `wait` volte sem que ninguém tenha notificado. **Notificação perdida**: se o `notify_one` acontecer antes de a outra thread começar a esperar, ela dormiria para sempre. Com o predicado, `wait` testa a condição antes de dormir e de novo a cada despertar, só voltando quando ela for verdadeira. A variável `pronto` é alterada com o mutex travado, o que garante que o teste e a espera não percam a mudança.

Exercícios de código — respostas

Q63 **Corrida de dados medida**

> *Escreva um programa com quatro threads, cada uma incrementando 100 mil vezes: (a) um `std::atomic<int>`; (b) um `int` protegido por `std::mutex`; (c) um contador local por thread, somado no fim. Os três devem dar 400000. (Não inclua a versão com `int` sem proteção: é comportamento indefinido.)*

```cpp
#include <atomic>
#include <mutex>
#include <print>
#include <thread>
#include <vector>

constexpr int N = 100'000;
constexpr int THREADS = 4;

int main() {
    std::atomic<int> atomico{0};
    int protegido = 0;
    std::mutex m;
    std::vector<int> parciais(THREADS);

    {
        std::vector<std::jthread> ts;
        for (int t = 0; t < THREADS; ++t) {
            ts.emplace_back([&, t] {
                for (int i = 0; i < N; ++i) {
                    ++atomico;
                    {
                        std::lock_guard lk(m);
                        ++protegido;
                    }
                    ++parciais[t];  // cada thread escreve só na sua posição
                }
            });
        }
    }  // os jthread fazem join aqui

    int soma = 0;
    for (int p : parciais)
        soma += p;
    std::println("atômico={} mutex={} local={}", atomico.load(), protegido, soma);
}
```

Saída

```
atômico=400000 mutex=400000 local=400000
```

O vetor `parciais` é compartilhado, mas cada thread escreve só no **seu** elemento: não há corrida. Ainda assim, elementos vizinhos ficam na mesma linha de cache, e as threads disputam essa linha (*false sharing*); somar numa variável local e gravar no vetor só no fim é mais rápido. Para conferir a versão sem proteção, compile com `-fsanitize=thread`: o ThreadSanitizer acusa a corrida mesmo quando o total sai certo.

Q64 Soma paralela com async

Some os números de 1 a 10 milhões dividindo o intervalo em quatro partes, cada uma calculada por um `std::async(std::launch::async, ...)`, e junte os `std::future<long long>`. Compare com a fórmula `n(n+1)/2`.

```cpp
#include <future>
#include <print>
#include <vector>

long long somar(long long de, long long ate) {
    long long s = 0;
    for (long long i = de; i <= ate; ++i)
        s += i;
    return s;
}

int main() {
    constexpr long long n = 10'000'000;
    constexpr int partes = 4;
    std::vector<std::future<long long>> fs;
    for (int p = 0; p < partes; ++p) {
        long long de = p * (n / partes) + 1;
        long long ate = (p == partes - 1) ? n : (p + 1) * (n / partes);
        fs.push_back(std::async(std::launch::async, somar, de, ate));
    }
    long long total = 0;
    for (auto& f : fs)
        total += f.get();  // espera e, se a tarefa lançou, relança aqui
    std::println("paralelo: {}", total);
    std::println("fórmula:  {}", n * (n + 1) / 2);
}
```

Saída

```
paralelo: 50000005000000
fórmula:  50000005000000
```

O `std::launch::async` força uma thread nova; sem ele, a implementação pode adiar a tarefa e executá-la só no `get()`, na thread de quem chama, sem paralelismo nenhum. O `future` também transporta exceções: se `somar` lançasse, o `get()` relançaria no `main`. A última parte pega o resto da divisão, para que nenhum número fique de fora quando `n` não divide por igual.

Q65 Produtor e consumidor

> *Escreva uma fila protegida por `std::mutex` e `std::condition_variable` (`push` e `pop` bloqueante) e use-a com um produtor que envia os números de 1 a 5 e depois um valor sentinela `-1`, e um consumidor que imprime o dobro de cada número até receber o sentinela.*

```cpp
#include <condition_variable>
#include <mutex>
#include <print>
#include <queue>
#include <thread>

template <typename T>
class FilaSegura {
public:
    void push(T v) {
        {
            std::lock_guard lk(m_);
            q_.push(std::move(v));
        }
        cv_.notify_one();  // fora da trava: quem acorda não precisa esperar por ela
    }
    T pop() {
        std::unique_lock lk(m_);
        cv_.wait(lk, [this] { return !q_.empty(); });
        T v = std::move(q_.front());
        q_.pop();
        return v;
    }

private:
    std::queue<T> q_;
    std::mutex m_;
    std::condition_variable cv_;
};

int main() {
    FilaSegura<int> fila;
    std::jthread consumidor([&] {
        for (int v; (v = fila.pop()) != -1;)
            std::println("consumiu {} -> {}", v, v * 2);
        std::println("consumidor encerrado");
    });
    for (int i = 1; i <= 5; ++i)
        fila.push(i);
    fila.push(-1);  // sentinela: "não vem mais nada"
}
```

Saída

```
consumiu 1 -> 2
consumiu 2 -> 4
consumiu 3 -> 6
consumiu 4 -> 8
consumiu 5 -> 10
consumidor encerrado
```

Com um só consumidor, a ordem de saída é determinística, porque a fila é FIFO. O `pop` usa `unique_lock`, não `lock_guard`, porque o `wait` precisa **destravar** o mutex enquanto dorme e travá-lo de novo ao acordar.

O sentinela é a forma mais simples de encerrar; com vários consumidores, envia-se um por consumidor ou usa-se um sinal de fechamento na própria fila.

MÓDULO 8

Ferramentas e qualidade

CMake, sanitizers, clang-tidy, testes com Catch2 e as Core Guidelines.

Artigos 35 a 38

#35 Do Comando Único ao Projeto Real — Construindo com CMake
#36 Redes de Segurança Automáticas — Sanitizers e clang-tidy
#37 Provando que Funciona — Testes Automatizados com Catch2
#38 A Sabedoria Codificada — Core Guidelines e Código Limpo em C++

Perguntas conceituais — respostas

Q66 Alvos e visibilidade no CMake

> *No CMake moderno, o que significam `PRIVATE`, `PUBLIC` e `INTERFACE` em `target_include_directories` e `target_link_libraries`?*

Eles dizem **quem herda** a configuração:

- `PRIVATE`: vale só para compilar o próprio alvo (um *include* usado apenas nos `.cpp` da biblioteca);
- `INTERFACE`: vale só para quem **usa** o alvo, não para ele (típico de bibliotecas só de cabeçalho);
- `PUBLIC`: as duas coisas (o cabeçalho público da biblioteca inclui algo dessa dependência).

Com isso, quem faz `target_link_libraries(app PRIVATE minidb)` recebe sozinho os *includes* e as dependências públicas de `minidb`, sem repetir caminhos. É a diferença entre o CMake "moderno", centrado em alvos, e o antigo, de variáveis globais como `include_directories`.

Q67 Qual sanitizer

> *Para cada bug, diga qual sanitizer o encontra: (a) ler `v[10]` num vetor de 10 elementos; (b) `INT_MAX + 1`; (c) duas threads escrevendo a mesma variável sem trava; (d) usar um `string_view` para uma string já destruída.*

- (a) **AddressSanitizer** (`-fsanitize=address`): *heap-buffer-overflow*.
- (b) **UndefinedBehaviorSanitizer** (`-fsanitize=undefined`): *signed integer overflow*.
- (c) **ThreadSanitizer** (`-fsanitize=thread`): *data race*. Não pode ser combinado com o ASan no mesmo executável.
- (d) **AddressSanitizer**: *heap-use-after-free* (ou *stack-use-after-return*, se a string era local e a opção estiver ativa).

Todos atuam **em execução**: só acham bugs nos caminhos que os testes percorrem. Por isso combinam com uma boa suíte de testes.

Q68 O que um teste unitário deve ter

> *Cite três características de um bom teste unitário e explique por que um teste que depende da ordem de execução dos outros é um problema.*

- **Rápido**: milissegundos, para rodar a cada mudança.
- **Isolado e determinístico**: o mesmo resultado sempre, sem depender de rede, relógio, ordem ou estado deixado por outros testes.
- **Verifica um comportamento**, com um nome que diz qual e uma falha que mostra o esperado e o obtido.

Um teste que depende de outro passa quando a suíte inteira roda e falha quando roda sozinho (ou quando o *framework* muda a ordem, como o Catch2 faz com `--order rand`). A falha aparece longe da causa, e o teste perde a utilidade.

Exercícios de código — respostas

Q69 Um CMakeLists para biblioteca e executável

Escreva um `CMakeLists.txt` para um projeto com uma biblioteca `calc` (em `src/calc.cpp`, com o cabeçalho público em `include/calc/calc.h`), um executável `app` (`src/main.cpp`) que a usa e um executável de testes `testes` ligado ao Catch2 baixado com `FetchContent`. Exija C++23, ative os avisos e permita ligar o AddressSanitizer com uma opção.

```
cmake_minimum_required(VERSION 3.25)
project(calc LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 23)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)  # -std=c++23, não -std=gnu++23

option(USAR_ASAN "Compila com AddressSanitizer e UBSan" OFF)
if(USAR_ASAN)
    add_compile_options(-fsanitize=address,undefined -fno-omit-frame-pointer)
    add_link_options(-fsanitize=address,undefined)
endif()

add_library(calc src/calc.cpp)
target_include_directories(calc PUBLIC include)  # quem usa calc enxerga include/
target_compile_options(calc PRIVATE -Wall -Wextra -Wpedantic)

add_executable(app src/main.cpp)
target_link_libraries(app PRIVATE calc)

include(FetchContent)
FetchContent_Declare(Catch2
    GIT_REPOSITORY https://github.com/catchorg/Catch2.git
    GIT_TAG v3.7.1)
FetchContent_MakeAvailable(Catch2)

add_executable(testes tests/test_calc.cpp)
target_link_libraries(testes PRIVATE calc Catch2::Catch2WithMain)

include(CTest)
include(Catch)
catch_discover_tests(testes)
```

Para compilar e testar: `cmake -B build -DUSAR_ASAN=ON && cmake --build build && ctest --test-dir build`. O diretório `build` fica fora do código-fonte e pode ser apagado a qualquer momento. O bloco do `USAR_ASAN` fica antes de todos os alvos de propósito: `add_compile_options` só vale para os alvos criados **depois** dele.

Q70 Testando sem framework

> *Antes do Catch2, escreva um mini-framework de testes com uma macro `VERIFICA(expr)` que registre a falha com arquivo, linha e o texto da expressão, sem interromper os outros testes. Use-o para testar uma função `bool eh_palindromo(std::string_view)` (incluindo um caso que falha de propósito) e devolva um código de saída diferente de zero se algum teste falhar.*

Aqui uma macro ainda é a ferramenta certa: só ela captura o **texto** da expressão (`#expr`), o arquivo e a linha do ponto de chamada.

```
#include <print>
#include <string_view>

static int falhas = 0, total = 0;

#define VERIFICA(expr)                                                      \
    do {                                                                    \
        ++total;                                                            \
        if (!(expr)) {                                                      \
            ++falhas;                                                       \
            std::println("FALHOU {}:{}: {}", __FILE_NAME__, __LINE__, #expr);     \
        }                                                                   \
    } while (0)

bool eh_palindromo(std::string_view s) {
    for (std::size_t i = 0, j = s.size(); i < j; ++i, --j)
        if (s[i] != s[j - 1])
            return false;
    return true;
}

int main() {
    VERIFICA(eh_palindromo(""));
    VERIFICA(eh_palindromo("a"));
    VERIFICA(eh_palindromo("arara"));
    VERIFICA(eh_palindromo("abba"));
    VERIFICA(!eh_palindromo("abca"));
    VERIFICA(eh_palindromo("Arara"));  // falha de propósito: compara maiúsculas
    std::println("{} de {} testes passaram", total - falhas, total);
    return falhas ? 1 : 0;
}
```

Saída

```
FALHOU main.cpp:28: eh_palindromo("Arara")
5 de 6 testes passaram
```

O teste com "Arara" mostra o valor de um teste que falha: ele documenta uma decisão (diferenciar maiúsculas) que talvez não fosse a desejada. O `__FILE_NAME__` (clang e gcc 12+) dá só o nome do arquivo; o `__FILE__` padrão traria o caminho completo. O Catch2 faz o mesmo com `REQUIRE` e `CHECK`, e ainda mostra os valores dos dois lados de uma comparação.

Q71 Seguindo as Core Guidelines

Reescreva este trecho seguindo as Core Guidelines e explique cada mudança com a regra correspondente (pelo tema, não é preciso decorar o número):

```
int* criar(int n) {
    int* v = (int*)malloc(n * sizeof(int));
    for (int i = 0; i <= n; i++) v[i] = 0;
    return v;
}
```

```
#include <print>
#include <vector>

std::vector<int> criar(std::size_t n) {
    return std::vector<int>(n, 0);
}

int main() {
    auto v = criar(5);
    std::println("tamanho {} | último {}", v.size(), v.back());
}
```

Saída

```
tamanho 5 | último 0
```

- **Posse explícita**: um `int*` cru devolvido não diz quem deve liberar, nem com `free` ou `delete`. O `std::vector` é o dono e libera sozinho (R.11: evitar `new` / `malloc` explícitos; R.3: ponteiro cru não tem posse).
- **Sem cast no estilo C**: `(int*)` esconde conversões perigosas (ES.48: evitar casts).
- **Sem erro de limite**: o `i <= n` original escreve uma posição além do fim. O construtor do vetor elimina o laço, e com ele o erro (ES.1: preferir a biblioteca padrão a código feito à mão).
- **Tipo do tamanho**: `std::size_t` em vez de `int`, porque um tamanho negativo não faz sentido.
- **Retorno por valor**: devolver o vetor não copia nada, graças à elisão de cópia garantida (F.20: preferir retornar valores a parâmetros de saída).

MÓDULO 9

Projeto final e extensões: o mini banco de dados, a web e a persistência

O mini banco de dados em memória (célula com variant, tabelas com RAII e STL, consultas com templates e lambdas, erros com expected) e as extensões: SQLite, HTTP, HTML gerado e dependências.

Artigos 39 a 52

#39 O Projeto Final Começa — Arquitetando um Mini Banco de Dados em Memória
#40 A Primeira Pedra — Uma Célula Poliforme com std::variant
#41 A Segunda Pedra — Linhas e Tabelas com RAII e STL
#42 A Terceira Pedra — Um Motor de Consultas com Templates e Lambdas
#43 A Quarta Pedra — Tratamento de Erros Coerente com std::expected
#44 A Pedra de Acabamento — Integração, Testes e Verificação do Sistema
#45 A Jornada Completa — Retrospectiva de Um Ano de C++
#46 Extensão — C++ no Mundo da Web e da Persistência
#47 Extensão — Dados que Sobrevivem: Integrando SQLite com RAII
#48 Extensão — O Terminal Vira Rede: Um Servidor HTTP Mínimo
#49 Extensão — Páginas Dinâmicas: HTML Gerado com Templates
#50 Extensão — Além do SQLite: PostgreSQL, MySQL e MongoDB
#51 Extensão — A Arte de Integrar: Dependências e Bibliotecas de Terceiros
#52 O Destino e o Novo Começo — Encerrando a Jornada Completa

Perguntas conceituais — respostas

Q72 variant ou herança para a célula

> *No projeto final, a célula da tabela (`Value`) guarda um inteiro, um real ou um texto num `std::variant`. Que vantagens isso tem sobre uma hierarquia `ValorBase` com `ValorInteiro`, `ValorReal` e `ValorTexto`? Quando a herança seria a melhor escolha?*

- **Valor, não ponteiro**: o `variant` guarda o dado no próprio objeto, sem alocação por célula e sem `unique_ptr`; uma linha com dez células é um bloco contíguo, e copiar uma célula é copiar um valor.
- **Conjunto fechado e verificado**: o `std::visit` obriga a tratar os três tipos, e esquecer um é erro de compilação. Com herança, uma operação nova (comparar, serializar) exige uma virtual nova em todas as classes.
- **Comparação e cópia simples**: `==` e cópia funcionam direto, sem clonagem polimórfica.

A herança seria melhor se o conjunto de tipos fosse **aberto**: se usuários do banco precisassem acrescentar tipos próprios (data, geometria) sem alterar o código da biblioteca. É o trade-off clássico: `variant` facilita acrescentar operações; herança facilita acrescentar tipos.

Q73 Onde ficam as invariantes da tabela

> *A tabela tem um esquema (nomes e tipos das colunas). Por que validar cada linha **na inserção**, e não na consulta? Que invariante isso garante?*

Validando na inserção, toda linha guardada **sempre** tem o número certo de células, cada uma do tipo da coluna. É a invariante da tabela: depois dela, o motor de consultas pode chamar `como_inteiro()` numa coluna inteira sem testar nada, porque um valor inválido nunca entrou. Validar na consulta espalharia a checagem por todos os leitores, repetiria o trabalho a cada leitura e só descobriria o problema longe de quem o causou. É o mesmo princípio da classe `Fracao` do módulo 2, aplicado a um contêiner.

Q74 Uma política de erros

> *O artigo #43 adota `std::expected` para erros de uso (coluna inexistente, tipo errado) e mantém exceções para falhas de programação ou de recurso (memória esgotada). Justifique a divisão.*

Erros de uso são **esperados**: quem chama o banco pode errar o nome de uma coluna, e deve poder reagir na hora (mostrar a mensagem, pedir outro valor). Com `expected`, a possibilidade de erro aparece no tipo de retorno, não custa caro e obriga quem chama a olhar o resultado. Já a falta de memória ou uma invariante interna violada não têm tratamento local razoável; a exceção sobe até um nível que possa registrar e encerrar, e o RAII limpa no caminho. Misturar as duas coisas, com exceções para tudo ou `expected` para tudo, encheria o código de `try` ou de checagens que ninguém sabe tratar.

Q75 RAII para uma API em C

> *A API do SQLite é em C: `sqlite3_open` e `sqlite3_close`, `sqlite3_prepare_v2` e `sqlite3_finalize`. Como o `std::unique_ptr` com um deleter próprio transforma esses pares em RAII, sem escrever uma classe completa?*

O `unique_ptr<T, D>` aceita um tipo `D` que diz **como** destruir. Com `struct FechaBanco { void operator()(sqlite3* db) const { sqlite3_close(db); } };`, um `std::unique_ptr<sqlite3, FechaBanco>` chama `sqlite3_close` no destrutor, e ainda é movível e não copiável, exatamente a semântica de posse que o *handle* exige. O mesmo vale para `sqlite3_stmt` com `sqlite3_finalize`. Como o *deleter* não tem estado, o ponteiro inteligente continua do tamanho de um ponteiro.

Q76 Um servidor HTTP mínimo

> *Um servidor HTTP feito à mão com sockets lê a requisição com `recv`. Por que é um erro assumir que uma chamada a `recv` traz a requisição inteira? E por que um servidor de estudo costuma responder com `Connection: close`?*

O TCP é um **fluxo de bytes**, sem fronteiras de mensagem: uma requisição pode chegar em vários pedaços (e duas requisições podem chegar juntas). O servidor precisa acumular os bytes até encontrar o fim do cabeçalho (`\r\n\r\n`) e, se houver corpo, ler mais `Content-Length` bytes. Assumir uma leitura só funciona no teste local e falha na rede real. O `Connection: close` diz ao cliente que a conexão termina com a resposta: o servidor não precisa gerenciar conexões persistentes (*keep-alive*), e o fim da resposta é o fim da conexão. É uma simplificação honesta para estudo, não para produção.

Q77 Bibliotecas de terceiros

Compare três formas de trazer uma biblioteca para um projeto C++: copiar o código para o repositório, `FetchContent` do CMake e um gerenciador de pacotes (vcpkg ou Conan). Qual usar num projeto de equipe?

- **Copiar** (*vendoring*): simples e funciona offline, mas atualizar é manual, e o histórico se mistura com código alheio.
- ** `FetchContent` *: baixa e compila a dependência junto com o projeto, numa versão fixada por *tag* ou *hash*. Ótimo para bibliotecas pequenas, como o Catch2; com dependências grandes (Boost, OpenSSL), cada compilação limpa recompila tudo.
- **vcpkg ou Conan**: resolvem versões e dependências transitivas, guardam binários prontos e integram com o CMake por um *toolchain file*. Exigem configurar a ferramenta.

Em equipe, um gerenciador de pacotes com versões fixadas num manifesto (`vcpkg.json` com *baseline*, ou o *lockfile* do Conan) dá *builds* reprodutíveis para todos. Para uma ou duas bibliotecas pequenas, o `FetchContent` com *tag* fixa basta.

Exercícios de código — respostas

Q78 Uma célula com variant e ordem entre tipos

Escreva `class Value` sobre `std::variant<std::int64_t, double, std::string>`, com `para_texto()` usando `std::visit` e um `operator<=>` próprio em que inteiros e reais se comparam pelo valor numérico (2 é menor que 2.5) e todo número vem antes de todo texto. Ordene um vetor misto.

```cpp
#include <algorithm>
#include <compare>
#include <cstdint>
#include <format>
#include <print>
#include <string>
#include <variant>
#include <vector>

class Value {
public:
    Value(std::int64_t v) : dado_(v) {}
    Value(double v) : dado_(v) {}
    Value(std::string v) : dado_(std::move(v)) {}
    Value(const char* v) : dado_(std::string(v)) {}

    bool eh_texto() const { return std::holds_alternative<std::string>(dado_); }
    double como_numero() const {
        return std::visit([](const auto& v) -> double {
            if constexpr (std::is_same_v<std::decay_t<decltype(v)>, std::string>)
                return 0;  // não chamado para texto: quem chama testa eh_texto antes
            else
                return static_cast<double>(v);
        }, dado_);
    }

    std::string para_texto() const {
        return std::visit([](const auto& v) { return std::format("{}", v); }, dado_);
    }

    std::partial_ordering operator<=>(const Value& o) const {
        if (eh_texto() != o.eh_texto())
            return eh_texto() ? std::partial_ordering::greater : std::partial_ordering::less;
        if (eh_texto())
            return std::get<std::string>(dado_) <=> std::get<std::string>(o.dado_);
        return como_numero() <=> o.como_numero();  // double: pode ser "unordered" (NaN)
    }
    bool operator==(const Value& o) const { return (*this <=> o) == 0; }

private:
    std::variant<std::int64_t, double, std::string> dado_;
};

int main() {
    std::vector<Value> vs{Value{"pera"}, Value{std::int64_t{3}}, Value{2.5}, Value{"abacate"},
                          Value{std::int64_t{2}}, Value{-1.0}};
    std::sort(vs.begin(), vs.end());
    for (const auto& v : vs)
        std::print("{} ", v.para_texto());
    std::println("");
    std::println("2 == 2.0? {}", Value{std::int64_t{2}} == Value{2.0});
}
```

Saída

```
-1 2 2.5 3 abacate pera
2 == 2.0? true
```

O tipo de retorno é `std::partial_ordering` porque `double` tem NaN, que não é menor, igual nem maior que nada. Para o `std::sort`, isso é um risco: com NaN no vetor, a ordem deixa de ser estrita e fraca e o comportamento é indefinido. O banco real deve recusar NaN na inserção (outra invariante). Comparar `int64_t` grandes convertendo para `double` perde precisão acima de 2^{53}; uma versão rigorosa compara inteiro com inteiro sem converter.

Q79 Tabela com esquema e expected

Escreva `struct Coluna { std::string nome; Tipo tipo; }` (com `enum class Tipo { Inteiro, Texto }`), `struct Erro { std::string mensagem; }` e `class Tabela` com `std::expected<void, Erro> inserir(std::vector<Valor> linha)`, que recuse linhas com número errado de células ou tipo errado. Use `using Valor = std::variant<long, std::string>;`. Mostre uma inserção válida e duas inválidas.

```
#include <cstddef>
#include <expected>
#include <format>
#include <print>
#include <string>
#include <variant>
#include <vector>

enum class Tipo { Inteiro, Texto };
using Valor = std::variant<long, std::string>;

struct Coluna {
    std::string nome;
    Tipo tipo;
};

struct Erro {
    std::string mensagem;
};

class Tabela {
public:
    explicit Tabela(std::vector<Coluna> esquema) : esquema_(std::move(esquema)) {}

    std::expected<void, Erro> inserir(std::vector<Valor> linha) {
        if (linha.size() != esquema_.size())
            return std::unexpected(Erro{std::format("esperava {} valores, recebeu {}",
esquema_.size(), linha.size())});
        for (std::size_t i = 0; i < linha.size(); ++i) {
            bool eh_inteiro = std::holds_alternative<long>(linha[i]);
            if (eh_inteiro != (esquema_[i].tipo == Tipo::Inteiro))
                return std::unexpected(Erro{std::format("coluna '{}': tipo errado",
esquema_[i].nome)});
        }
        linhas_.push_back(std::move(linha));
        return {};
    }
    std::size_t tamanho() const { return linhas_.size(); }

private:
    std::vector<Coluna> esquema_;
    std::vector<std::vector<Valor>> linhas_;
};

int main() {
    Tabela livros({{"titulo", Tipo::Texto}, {"ano", Tipo::Inteiro}});
    std::vector<std::vector<Valor>> tentativas{
        {std::string("Vidas Secas"), 1938L},
        {std::string("Dom Casmurro")},
        {1899L, std::string("Dom Casmurro")},
    };
    for (auto& t : tentativas) {
        if (auto r = livros.inserir(std::move(t)); r)
```

```
            std::println("ok");
        else
            std::println("recusada: {}", r.error().mensagem);
    }
    std::println("linhas guardadas: {}", livros.tamanho());
}
```

Saída

```
ok
recusada: esperava 2 valores, recebeu 1
recusada: coluna 'titulo': tipo errado
linhas guardadas: 1
```

`std::expected<void, Erro>` é o jeito de dizer "não devolve nada, mas pode falhar": o sucesso é um `return {};`. A linha é recebida por valor e movida para dentro da tabela: quem chama decide se copia ou entrega.

Q80 Um motor de consultas fluente

Escreva `template <typename T> class Consulta` que receba um `std::vector<T>` e ofereça `onde(pred)`, `ordenar_por(projecao)` e `limite(n)`, cada um devolvendo `Consulta&` para encadear, e `executar()` que devolve o resultado. Use-a para listar os dois livros mais antigos publicados depois de 1900.

```
#include <algorithm>
#include <cstddef>
#include <functional>
#include <print>
#include <string>
#include <vector>

template <typename T>
class Consulta {
public:
    explicit Consulta(const std::vector<T>& dados) : dados_(dados) {}

    Consulta& onde(std::function<bool(const T&)> p) {
        filtros_.push_back(std::move(p));
        return *this;
    }
    template <typename Proj>
    Consulta& ordenar_por(Proj proj) {
        ordem_ = [proj](const T& a, const T& b) { return std::invoke(proj, a) < std::invoke(proj,
b); };
        return *this;
    }
    Consulta& limite(std::size_t n) {
        limite_ = n;
        return *this;
    }

    std::vector<T> executar() const {
        std::vector<T> r;
        std::copy_if(dados_.begin(), dados_.end(), std::back_inserter(r), [this](const T& x) {
            return std::all_of(filtros_.begin(), filtros_.end(), [&x](const auto& f) { return
f(x); });
        });
        if (ordem_)
            std::stable_sort(r.begin(), r.end(), ordem_);
        if (r.size() > limite_)
            r.resize(limite_);
        return r;
    }

private:
    const std::vector<T>& dados_;
    std::vector<std::function<bool(const T&)>> filtros_;
    std::function<bool(const T&, const T&)> ordem_;
    std::size_t limite_ = static_cast<std::size_t>(-1);
};

struct Livro {
    std::string titulo;
    int ano;
};

int main() {
```

```
    std::vector<Livro> acervo{{"Vidas Secas", 1938}, {"Dom Casmurro", 1899}, {"Macunaíma", 1928},
                              {"Grande Sertão: Veredas", 1956}, {"O Cortiço", 1890}};
    auto r = Consulta(acervo)
                .onde([](const Livro& l) { return l.ano > 1900; })
                .ordenar_por(&Livro::ano)
                .limite(2)
                .executar();
    for (const auto& l : r)
        std::println("{} ({})", l.titulo, l.ano);
}
```

Saída

```
Macunaíma (1928)
Vidas Secas (1938)
```

`ordenar_por(&Livro::ano)` funciona porque o `std::invoke` aceita um ponteiro para membro como se fosse uma função que recebe o objeto. O filtro só roda no `executar`: é uma consulta **preguiçosa**, como as views de `std::ranges`. Guardar `dados_` por referência é seguro só enquanto o vetor original viver mais que a consulta; uma consulta devolvida para fora do escopo do vetor seria um *dangling reference*.

Q81 SQLite com RAII e parâmetros

Usando a API C do SQLite (`<sqlite3.h>`, compile com `-lsqlite3`), crie um banco em memória com a tabela `livros(titulo, ano)`, insira três livros com um statement preparado e parâmetros `?` (um dos títulos deve ser uma tentativa de injeção de SQL) e liste-os em ordem de ano. Administre o banco e os statements com `std::unique_ptr` e deleters próprios.

```
#include <sqlite3.h>

#include <memory>
#include <print>
#include <stdexcept>
#include <utility>

struct FechaBanco {
    void operator()(sqlite3* db) const { sqlite3_close(db); }
};
struct FinalizaStmt {
    void operator()(sqlite3_stmt* s) const { sqlite3_finalize(s); }
};
using Banco = std::unique_ptr<sqlite3, FechaBanco>;
using Stmt = std::unique_ptr<sqlite3_stmt, FinalizaStmt>;

Banco abrir(const char* caminho) {
    sqlite3* bruto = nullptr;
    int rc = sqlite3_open(caminho, &bruto);
    Banco db(bruto);  // mesmo quando falha, o sqlite3_open devolve um handle que precisa ser
fechado
    if (rc != SQLITE_OK)
        throw std::runtime_error(sqlite3_errmsg(bruto));
    return db;
}

Stmt preparar(sqlite3* db, const char* sql) {
    sqlite3_stmt* s = nullptr;
    if (sqlite3_prepare_v2(db, sql, -1, &s, nullptr) != SQLITE_OK)
        throw std::runtime_error(sqlite3_errmsg(db));
    return Stmt(s);
}

void passo(sqlite3* db, sqlite3_stmt* s) {
    if (sqlite3_step(s) != SQLITE_DONE)
        throw std::runtime_error(sqlite3_errmsg(db));
}

int main() {
    try {
        auto db = abrir(":memory:");
        passo(db.get(), preparar(db.get(), "CREATE TABLE livros (titulo TEXT NOT NULL, ano
INTEGER)").get());

        auto ins = preparar(db.get(), "INSERT INTO livros (titulo, ano) VALUES (?, ?)");
        for (auto [t, a] : {std::pair{"Dom Casmurro", 1899}, {"Vidas Secas", 1938},
                            {"x'); DROP TABLE livros; --", 2024}}) {
            sqlite3_bind_text(ins.get(), 1, t, -1, SQLITE_TRANSIENT);
            sqlite3_bind_int(ins.get(), 2, a);
            passo(db.get(), ins.get());
            sqlite3_reset(ins.get());  // reaproveita o statement preparado
        }
```

```
        auto sel = preparar(db.get(), "SELECT titulo, ano FROM livros ORDER BY ano");
        while (sqlite3_step(sel.get()) == SQLITE_ROW) {
            auto titulo = reinterpret_cast<const char*>(sqlite3_column_text(sel.get(), 0));
            std::println("{} ({})", titulo, sqlite3_column_int(sel.get(), 1));
        }
    } catch (const std::exception& e) {
        std::println("erro do banco: {}", e.what());
        return 1;
    }
}
```

Saída

```
Dom Casmurro (1899)
Vidas Secas (1938)
x'); DROP TABLE livros; -- (2024)
```

O título malicioso foi guardado **como texto**, e a tabela continua existindo: com parâmetros, o valor nunca é interpretado como SQL. Se o código montasse a string com `"... VALUES ('" + titulo + "', ...)"`, o `'` fecharia o texto e o resto viraria comando. A ordem de destruição também importa: os *statements* são finalizados antes do banco fechar porque foram criados depois, e variáveis locais morrem na ordem inversa da criação.

Q82 Analisando a linha de requisição HTTP

Sem sockets, escreva `std::expected<Requisicao, std::string> analisar(std::string_view linha)` que separe uma linha como `GET /livros?autor=ramos HTTP/1.1` em método, caminho, query string e versão. Recuse métodos fora de GET, POST, PUT e DELETE, caminhos que não começam com `/` e versões que não começam com `HTTP/`.

```
#include <array>
#include <algorithm>
#include <expected>
#include <print>
#include <string>
#include <string_view>

struct Requisicao {
    std::string metodo, caminho, query, versao;
};

std::expected<Requisicao, std::string> analisar(std::string_view linha) {
    auto e1 = linha.find(' ');
    auto e2 = linha.find(' ', e1 == linha.npos ? e1 : e1 + 1);
    if (e1 == linha.npos || e2 == linha.npos || linha.find(' ', e2 + 1) != linha.npos)
        return std::unexpected("formato: MÉTODO CAMINHO VERSÃO");

    std::string_view metodo = linha.substr(0, e1);
    std::string_view alvo = linha.substr(e1 + 1, e2 - e1 - 1);
    std::string_view versao = linha.substr(e2 + 1);

    constexpr std::array<std::string_view, 4> aceitos{"GET", "POST", "PUT", "DELETE"};
    if (std::ranges::find(aceitos, metodo) == aceitos.end())
        return std::unexpected("método não suportado: " + std::string(metodo));
    if (!alvo.starts_with('/'))
        return std::unexpected("caminho deve começar com /");
    if (!versao.starts_with("HTTP/"))
        return std::unexpected("versão inválida");

    auto q = alvo.find('?');
    return Requisicao{std::string(metodo), std::string(alvo.substr(0, q)),
                      q == alvo.npos ? "" : std::string(alvo.substr(q + 1)),
std::string(versao)};
}

int main() {
    for (std::string_view l : {"GET /livros?autor=ramos HTTP/1.1", "POST /livros HTTP/1.1",
                               "PATCH /livros HTTP/1.1", "GET livros HTTP/1.1", "GET /"}) {
        if (auto r = analisar(l))
            std::println("{} | caminho={} query={} versao={}", r->metodo, r->caminho,
                         r->query.empty() ? "-" : r->query, r->versao);
        else
            std::println("400 Bad Request: {}", r.error());
    }
}
```

Saída

```
GET | caminho=/livros query=autor=ramos versao=HTTP/1.1
POST | caminho=/livros query=- versao=HTTP/1.1
400 Bad Request: método não suportado: PATCH
400 Bad Request: caminho deve começar com /
400 Bad Request: formato: MÉTODO CAMINHO VERSÃO
```

Separar a análise do *socket* torna o código testável sem rede. Um servidor real ainda precisaria decodificar `%20` e afins na *query*, limitar o tamanho da linha (para não aceitar 1 GB de cabeçalho) e tratar o `\r\n` do fim, que aqui já chega removido.

Q83 HTML gerado com segurança

Escreva `std::string escapar(std::string_view)` que troque `& < > " '` pelas entidades HTML e `std::string renderizar(std::string_view modelo, const std::map<std::string, std::string>& vars)` que substitua `{{nome}}` pelo valor escapado. Mostre o resultado com um nome que contém `<script>`.

```
#include <map>
#include <print>
#include <string>
#include <string_view>

std::string escapar(std::string_view s) {
    std::string r;
    r.reserve(s.size());
    for (char c : s) {
        switch (c) {
        case '&': r += "&amp;"; break;
        case '<': r += "&lt;"; break;
        case '>': r += "&gt;"; break;
        case '"': r += "&quot;"; break;
        case '\'': r += "&#39;"; break;
        default: r += c;
        }
    }
    return r;
}

std::string renderizar(std::string_view modelo, const std::map<std::string, std::string>& vars) {
    std::string r;
    std::size_t i = 0;
    while (i < modelo.size()) {
        auto ini = modelo.find("{{", i);
        auto fim = ini == modelo.npos ? modelo.npos : modelo.find("}}", ini + 2);
        if (fim == modelo.npos) {
            r += modelo.substr(i);
            break;
        }
        r += modelo.substr(i, ini - i);
        std::string nome(modelo.substr(ini + 2, fim - ini - 2));
        auto it = vars.find(nome);
        r += it != vars.end() ? escapar(it->second) : "";  // variável ausente vira vazio
        i = fim + 2;
    }
    return r;
}

int main() {
    std::string modelo = "<p>Olá, {{nome}}! Você tem {{n}} livros.</p>";
    std::println("{}", renderizar(modelo, {{"nome", "Ana"}, {"n", "3"}}));
    std::println("{}", renderizar(modelo, {{"nome", "<script>alert('x')</script>"}, {"n",
"0"}}));
}
```

Saída

```
<p>Olá, Ana! Você tem 3 livros.</p>
<p>Olá, &lt;script&gt;alert(&#39;x&#39;)&lt;/script&gt;! Você tem 0 livros.</p>
```

Escapar **na saída**, no momento de montar o HTML, é a defesa certa contra XSS: o mesmo texto pode ir para HTML, JSON ou SQL, e cada destino tem as próprias regras. Escapar só dentro de atributos exigiria mais cuidado (atributos sem aspas, URLs com `javascript:`); por isso os motores de template de verdade escapam por contexto.

MÓDULO 10

Jogos

O game loop, raylib, o padrão ECS, colisões, recursos com RAII e estados de jogo: Pong, Snake e Breakout.

Artigos 53 a 59

#53 Um Novo Território — Por que Jogos, o Game Loop e as Três Bibliotecas
#54 O Primeiro Pixel — Janela, Desenho e Input com raylib
#55 Entidades de Verdade — Componentes e o Padrão ECS
#56 Quando as Coisas se Tocam — Colisão e o Primeiro Jogo: Pong
#57 Imagem e Som — Recursos de Jogo com RAII, e o Início do Snake
#58 Menu, Jogo, Fim — Estados de Jogo e o Snake Completo
#59 O Capstone dos Jogos — Breakout e o Fim da Jornada

Perguntas conceituais — respostas

Q84 O game loop

Descreva as três etapas de um game loop e explique por que usar o tempo do quadro (delta time) no movimento. Que problema o passo fixo resolve que o delta time variável não resolve?

A cada quadro: **ler a entrada**, **atualizar** o estado do jogo e **desenhar**. Se a posição andar um valor fixo por quadro, o jogo fica duas vezes mais rápido num monitor de 120 Hz que num de 60 Hz; multiplicando a velocidade pelo *delta time* (`x += v * dt`), a velocidade passa a ser por segundo, independente da taxa de quadros. Mas um `dt` variável deixa a física **não determinística** e instável: um quadro lento produz um `dt` grande, e um objeto rápido atravessa uma parede sem nunca a tocar (*tunneling*). O **passo fixo** acumula o tempo real e roda a física em fatias iguais (por exemplo, 1/60 s), quantas couberem; o desenho continua no ritmo do monitor.

Q85 ECS ou herança

Num jogo, por que uma hierarquia como `Entidade → Movel → Inimigo → InimigoVoador` tende a se complicar? Como o padrão ECS (Entity Component System) organiza o mesmo jogo?

A herança obriga a escolher **um** eixo de classificação. Quando surge um inimigo que voa e também pode ser pilotado, ou uma moeda que se move, a hierarquia precisa de herança múltipla, de classes duplicadas ou de virtuais vazias na base. No ECS, a **entidade** é só um identificador; os **componentes** são dados simples (`Posicao`, `Velocidade`, `Vida`, `Aparencia`), ligados a ela livremente; e os **sistemas** são funções que processam todas as entidades com um conjunto de componentes (o sistema de movimento atua em quem tem `Posicao` e `Velocidade`). Comportamento novo é uma combinação nova de componentes, não uma classe nova. De quebra, componentes iguais guardados juntos em vetores aproveitam bem o cache.

Q86 Recursos de jogo e a ordem de destruição

Uma classe `Textura` com RAII chama `LoadTexture` no construtor e `UnloadTexture` no destrutor. Por que declarar uma `Textura` global quebra o programa, e como organizar a vida dos recursos em relação a `InitWindow` e `CloseWindow`?

`LoadTexture` exige o contexto gráfico (OpenGL) que o `InitWindow` cria. Uma `Textura` global é construída **antes** do `main`, portanto antes da janela existir, e é destruída **depois** do fim do `main`, depois do `CloseWindow`: tanto a carga quanto a liberação acontecem sem contexto. A solução é dar à janela um objeto RAII também (`Janela` chama `InitWindow` no construtor e `CloseWindow` no destrutor) e declarar os recursos **depois** dela, no mesmo escopo. A ordem inversa de destruição garante que as texturas sejam liberadas antes de a janela fechar.

Exercícios de código — respostas

Q87 Passo fixo na prática

> *Simule um game loop sem gráficos: os quadros chegam com tempos de 0,016 s, 0,033 s, 0,008 s, 0,050 s e 0,016 s. A física roda em passos fixos de 0,02 s, com um objeto a 100 unidades por segundo. Para cada quadro, mostre quantos passos de física rodaram, a posição e o tempo que sobrou no acumulador.*

```
#include <array>
#include <print>

int main() {
    constexpr double passo = 0.02;
    constexpr double velocidade = 100.0;
    constexpr std::array quadros{0.016, 0.033, 0.008, 0.050, 0.016};

    double acumulador = 0, x = 0;
    for (double dt : quadros) {
        acumulador += dt;
        int passos = 0;
        while (acumulador >= passo) {  // a física sempre avança em fatias iguais
            x += velocidade * passo;
            acumulador -= passo;
            ++passos;
        }
        std::println("quadro {:.3f}s: {} passo(s), x = {:4.1f}, sobra {:.3f}s", dt, passos, x,
acumulador);
    }
}
```

Saída

```
quadro 0.016s: 0 passo(s), x =  0.0, sobra 0.016s
quadro 0.033s: 2 passo(s), x =  4.0, sobra 0.009s
quadro 0.008s: 0 passo(s), x =  4.0, sobra 0.017s
quadro 0.050s: 3 passo(s), x = 10.0, sobra 0.007s
quadro 0.016s: 1 passo(s), x = 12.0, sobra 0.003s
```

Repare no primeiro quadro, sem nenhum passo, e no quarto, com três: a física não acompanha cada quadro, mas no total ela avança exatamente o tempo decorrido (0,123 s ≈ 6 passos e a sobra). Para o desenho não "travar", jogos interpolam a posição entre os dois últimos estados usando `sobra / passo` como fração. Um teto para o número de passos por quadro evita a "espiral da morte", quando a física fica mais lenta que o tempo real.

Q88 Colisão AABB e o rebote do Pong

Escreva `struct Retangulo { float x, y, w, h; }` e `bool colide(const Retangulo&, const Retangulo&)` (caixas alinhadas aos eixos). Simule uma bola de Pong que anda 4 unidades por passo em x e 1 em y e rebate numa raquete em x = 20: ao colidir, inverta a velocidade horizontal e empurre a bola para fora da raquete. Mostre os passos em que houve colisão.

```
#include <print>

struct Retangulo {
    float x, y, w, h;
};

bool colide(const Retangulo& a, const Retangulo& b) {
    return a.x < b.x + b.w && b.x < a.x + a.w && a.y < b.y + b.h && b.y < a.y + a.h;
}

int main() {
    Retangulo raquete{20, 0, 2, 40};
    Retangulo bola{2, 10, 3, 3};
    float vx = 4, vy = 1;
    for (int passo = 1; passo <= 10; ++passo) {
        bola.x += vx;
        bola.y += vy;
        if (colide(bola, raquete)) {
            vx = -vx;
            bola.x = raquete.x - bola.w;  // sem isso, a bola fica "presa" dentro da raquete
            std::println("passo {}: colidiu, bola volta de x = {}", passo, bola.x);
        }
    }
    std::println("posição final: ({}, {}), vx = {}", bola.x, bola.y, vx);
}
```

Saída

```
passo 4: colidiu, bola volta de x = 17
posição final: (-7, 20), vx = -4
```

Sem empurrar a bola para fora, ela ainda estaria sobreposta à raquete no passo seguinte, e a velocidade seria invertida de novo: a bola "treme" e atravessa. A colisão por sobreposição também falha quando a bola anda mais que a largura da raquete num passo (*tunneling*); a defesa é o passo fixo pequeno ou um teste de varredura (*swept AABB*).

Q89 Um ECS mínimo

Escreva um ECS em que as entidades são índices, os componentes `Posicao`, `Velocidade` e `Vida` ficam em `std::vector<std::optional<T>>`, e dois sistemas: `mover` (atua em quem tem posição e velocidade) e `envenenar` (tira 1 de vida por passo e marca para remoção quem chega a zero). Crie um jogador, uma moeda parada e um inimigo envenenado, e rode três passos.

```
#include <cstddef>
#include <optional>
#include <print>
#include <string>
#include <vector>

struct Posicao { float x, y; };
struct Velocidade { float dx, dy; };
struct Vida { int pontos; };

struct Mundo {
    std::vector<std::string> nome;
    std::vector<std::optional<Posicao>> pos;
    std::vector<std::optional<Velocidade>> vel;
    std::vector<std::optional<Vida>> vida;
    std::vector<bool> viva;

    std::size_t criar(std::string n) {
        nome.push_back(std::move(n));
        pos.emplace_back();
        vel.emplace_back();
        vida.emplace_back();
        viva.push_back(true);
        return nome.size() - 1;
    }
};

void mover(Mundo& m) {
    for (std::size_t e = 0; e < m.nome.size(); ++e)
        if (m.viva[e] && m.pos[e] && m.vel[e]) {
            m.pos[e]->x += m.vel[e]->dx;
            m.pos[e]->y += m.vel[e]->dy;
        }
}

void envenenar(Mundo& m) {
    for (std::size_t e = 0; e < m.nome.size(); ++e)
        if (m.viva[e] && m.vida[e] && --m.vida[e]->pontos <= 0) {
            m.viva[e] = false;
            std::println("  {} morreu", m.nome[e]);
        }
}

int main() {
    Mundo m;
    auto jogador = m.criar("jogador");
    m.pos[jogador] = Posicao{0, 0};
    m.vel[jogador] = Velocidade{1, 0.5f};

    auto moeda = m.criar("moeda");
    m.pos[moeda] = Posicao{5, 5};  // tem posição, mas não velocidade: o sistema mover a ignora

    auto inimigo = m.criar("inimigo");
```

```
    m.pos[inimigo] = Posicao{10, 0};
    m.vel[inimigo] = Velocidade{-2, 0};
    m.vida[inimigo] = Vida{2};

    for (int passo = 1; passo <= 3; ++passo) {
        std::println("passo {}", passo);
        mover(m);
        envenenar(m);
        for (std::size_t e = 0; e < m.nome.size(); ++e)
            if (m.viva[e])
                std::println("  {} em ({}, {})", m.nome[e], m.pos[e]->x, m.pos[e]->y);
    }
}
```

Saída

```
passo 1
  jogador em (1, 0.5)
  moeda em (5, 5)
  inimigo em (8, 0)
passo 2
  inimigo morreu
  jogador em (2, 1)
  moeda em (5, 5)
passo 3
  jogador em (3, 1.5)
  moeda em (5, 5)
```

O jogador nunca é envenenado porque não tem `Vida`; a moeda nunca se move porque não tem `Velocidade`. Nenhuma das três entidades tem classe própria. ECS de produção (como o EnTT) usam vetores densos por componente, em vez de `optional` em todas as posições, para que cada sistema percorra só quem tem os componentes.

Q90 A cobra do Snake

Represente a cobra do Snake como um `std::deque<Ponto>` (a cabeça na frente). Escreva `bool andar(Direcao d, bool comeu)` que acrescente a nova cabeça, remova a cauda quando a cobra não comeu e devolva `false` se a cabeça bater no próprio corpo. Simule uma sequência de movimentos que termina em colisão.

```
#include <algorithm>
#include <deque>
#include <iterator>
#include <print>

struct Ponto {
    int x, y;
    bool operator==(const Ponto&) const = default;
};
enum class Direcao { Cima, Baixo, Esquerda, Direita };

class Cobra {
public:
    explicit Cobra(Ponto inicio) { corpo_.push_back(inicio); }

    bool andar(Direcao d, bool comeu) {
        Ponto c = corpo_.front();
        switch (d) {
        case Direcao::Cima: --c.y; break;
        case Direcao::Baixo: ++c.y; break;
        case Direcao::Esquerda: --c.x; break;
        case Direcao::Direita: ++c.x; break;
        }
        // se não comeu, a cauda vai sair neste passo: a cabeça pode ocupar a casa dela
        auto fim = comeu ? corpo_.end() : std::prev(corpo_.end());
        if (std::find(corpo_.begin(), fim, c) != fim)
            return false;
        if (!comeu)
            corpo_.pop_back();
        corpo_.push_front(c);
        return true;
    }
    std::size_t tamanho() const { return corpo_.size(); }

private:
    std::deque<Ponto> corpo_;
};

int main() {
    using enum Direcao;
    Cobra cobra({5, 5});
    struct Mov { Direcao d; bool comeu; const char* nome; };
    Mov movs[] = {{Direita, true, "direita (comeu)"}, {Direita, true, "direita (comeu)"},
                  {Direita, true, "direita (comeu)"}, {Direita, true, "direita (comeu)"},
                  {Baixo, false, "baixo"},
                  {Esquerda, false, "esquerda"}, {Cima, false, "cima"}};
    for (const auto& m : movs) {
        bool viva = cobra.andar(m.d, m.comeu);
        std::println("{:<16} tamanho {} {}", m.nome, cobra.tamanho(), viva ? "" : "-> bateu em si
mesma!");
        if (!viva)
            break;
```

```
    }
}
```

Saída

```
direita (comeu)  tamanho 2
direita (comeu)  tamanho 3
direita (comeu)  tamanho 4
direita (comeu)  tamanho 5
baixo            tamanho 5
esquerda         tamanho 5
cima             tamanho 5 -> bateu em si mesma!
```

Com quatro segmentos, a mesma volta (baixo, esquerda, cima) **não** seria colisão: a cabeça entraria na casa que a cauda acabou de deixar. Por isso o teste ignora a cauda quando a cobra não comeu; comparando com o corpo inteiro, ela morreria num movimento legal.

O `deque` insere e remove nas duas pontas em O(1), exatamente o que a cobra faz a cada passo; um `vector` precisaria deslocar o corpo todo para inserir a cabeça na frente. O `using enum` (C++20) evita repetir `Direcao::` na tabela de movimentos.

Q91 Estados de jogo

Modele os estados `Menu`, `Jogando`, `Pausado` e `FimDeJogo` com um `enum class` e uma função `Estado proximo(Estado atual, Evento e)` com as transições: Enter no menu começa o jogo; P pausa e despausa; perder vida leva ao fim de jogo; Enter no fim volta ao menu. Eventos que não se aplicam ao estado atual são ignorados. Rode uma sequência de eventos e mostre cada transição.

```
#include <print>
#include <string_view>

enum class Estado { Menu, Jogando, Pausado, FimDeJogo };
enum class Evento { Enter, P, PerdeuVida };

std::string_view nome(Estado e) {
    switch (e) {
    case Estado::Menu: return "Menu";
    case Estado::Jogando: return "Jogando";
    case Estado::Pausado: return "Pausado";
    case Estado::FimDeJogo: return "FimDeJogo";
    }
    return "?";
}

Estado proximo(Estado atual, Evento ev) {
    switch (atual) {
    case Estado::Menu:
        return ev == Evento::Enter ? Estado::Jogando : atual;
    case Estado::Jogando:
        if (ev == Evento::P)
            return Estado::Pausado;
        return ev == Evento::PerdeuVida ? Estado::FimDeJogo : atual;
    case Estado::Pausado:
        return ev == Evento::P ? Estado::Jogando : atual;  // pausado, o jogo não perde vida
    case Estado::FimDeJogo:
        return ev == Evento::Enter ? Estado::Menu : atual;
    }
    return atual;
}

int main() {
    using enum Evento;
    Estado e = Estado::Menu;
    for (Evento ev : {P, Enter, P, PerdeuVida, P, PerdeuVida, Enter, Enter}) {
        Estado novo = proximo(e, ev);
        std::println("{:<9} -> {}", nome(e), novo == e ? "(ignorado)" : nome(novo));
        e = novo;
    }
}
```

Saída

```
Menu      -> (ignorado)
Menu      -> Jogando
Jogando   -> Pausado
Pausado   -> (ignorado)
Pausado   -> Jogando
Jogando   -> FimDeJogo
FimDeJogo -> Menu
Menu      -> Jogando
```

Concentrar as transições numa função pura torna as regras do jogo testáveis sem janela, sem teclado e sem relógio. No laço do jogo, cada estado tem também o seu `atualizar` e o seu `desenhar`, escolhidos por um `switch` sobre o estado atual. Com `-Wall`, o compilador avisa se um `switch` sobre o `enum class` esquecer um dos estados.

PARA IR ALÉM

Desafios integradores

Projetos maiores, que juntam vários módulos. Não há uma única resposta certa: o gabarito mostra um caminho.

D1

Uma calculadora de terminal

Escreva uma calculadora de terminal com sete operações: soma, subtração, multiplicação, divisão, potência (expoente inteiro), porcentagem (`15 % 200` é 15% de 200) e raiz quadrada (`raiz 2`). Ela deve: (a) aceitar vírgula ou ponto como separador decimal; (b) guardar o último resultado em `ans`, utilizável na conta seguinte; (c) recusar divisão por zero, raiz de negativo, expoente fracionário, resultado que estoura o `double` e texto que não é número com uma mensagem clara, ***sem encerrar o programa****; (d) ter os comandos `hist` (histórico) e `sair`. Separe o núcleo (`calc.h` e `calc.cpp`, sem entrada nem saída) da interface (`main.cpp`) e use `std::expected` para os erros.*

Caminho da solução

- **Núcleo**: a classe `Calculadora` guarda o `ans` e expõe `avaliar(linha)`, que devolve `std::expected<double, Erro>`. Nenhum `throw` e nenhuma saída no núcleo: ele pode ser usado por um terminal, por uma janela ou por um teste automatizado.
- **Números**: `std::from_chars` converte sem depender do *locale* e informa onde parou; a vírgula é trocada por ponto numa cópia do texto. O resultado é conferido com `std::isfinite`, para pegar o `2 ^ 2000`.
- **Interface**: lê linhas com `std::getline`, guarda o histórico num `std::vector<std::string>` e mostra os resultados com vírgula decimal.

`calc.h`:

```
#pragma once

#include <expected>
#include <string_view>

namespace calc {

enum class Erro { Formato, Numero, Operacao, DivisaoPorZero, RaizNegativa, Expoente, Estouro };

std::string_view mensagem(Erro e);

class Calculadora {
public:
    // Avalia "12,5 + 3", "2 ^ 10", "15 % 200" ou "raiz 2"; "ans" é o último resultado.
    std::expected<double, Erro> avaliar(std::string_view linha);
    double ans() const { return ans_; }

private:
    std::expected<double, Erro> numero(std::string_view texto) const;
    double ans_ = 0;
};

}  // namespace calc
```

`calc.cpp`:

```
#include "calc.h"

#include <charconv>
#include <cmath>
#include <string>
#include <vector>

namespace calc {

std::string_view mensagem(Erro e) {
    switch (e) {
    case Erro::Formato: return "use \"a op b\" ou \"raiz a\"";
    case Erro::Numero: return "operando não é um número";
    case Erro::Operacao: return "operação desconhecida (use + - * / ^ % ou raiz)";
    case Erro::DivisaoPorZero: return "divisão por zero";
    case Erro::RaizNegativa: return "raiz de número negativo";
    case Erro::Expoente: return "o expoente precisa ser inteiro";
    case Erro::Estouro: return "resultado grande demais";
    }
    return "erro desconhecido";
}

static std::vector<std::string_view> palavras(std::string_view s) {
    std::vector<std::string_view> r;
    std::size_t i = 0;
    while (i < s.size()) {
        while (i < s.size() && (s[i] == ' ' || s[i] == '\t'))
            ++i;
        std::size_t ini = i;
        while (i < s.size() && s[i] != ' ' && s[i] != '\t')
            ++i;
        if (i > ini)
            r.push_back(s.substr(ini, i - ini));
    }
    return r;
}

std::expected<double, Erro> Calculadora::numero(std::string_view texto) const {
    if (texto == "ans")
        return ans_;
    std::string copia(texto);
    for (char& c : copia)
        if (c == ',')
            c = '.';  // from_chars só entende ponto
    double v{};
    auto [fim, ec] = std::from_chars(copia.data(), copia.data() + copia.size(), v);
    if (ec != std::errc{} || fim != copia.data() + copia.size() || !std::isfinite(v))
        return std::unexpected(Erro::Numero);
    return v;
}
```

```
static std::expected<double, Erro> operar(std::string_view op, double a, double b) {
    if (op == "+") return a + b;
    if (op == "-") return a - b;
    if (op == "*") return a * b;
    if (op == "/") {
        if (b == 0)
            return std::unexpected(Erro::DivisaoPorZero);
        return a / b;
    }
    if (op == "^") {
        if (b != std::trunc(b))
            return std::unexpected(Erro::Expoente);
        return std::pow(a, b);
    }
    if (op == "%") return a * b / 100;
    return std::unexpected(Erro::Operacao);
}

std::expected<double, Erro> Calculadora::avaliar(std::string_view linha) {
    auto p = palavras(linha);
    std::expected<double, Erro> r = std::unexpected(Erro::Formato);
    if (p.size() == 2 && p[0] == "raiz") {
        r = numero(p[1]).and_then([](double x) -> std::expected<double, Erro> {
            if (x < 0)
                return std::unexpected(Erro::RaizNegativa);
            return std::sqrt(x);
        });
    } else if (p.size() == 3) {
        r = numero(p[0]).and_then([&](double a) {
            return numero(p[2]).and_then([&](double b) { return operar(p[1], a, b); });
        });
    }
    if (r && !std::isfinite(*r))
        return std::unexpected(Erro::Estouro);
    if (r)
        ans_ = *r;  // só um resultado válido substitui o ans
    return r;
}

}  // namespace calc
```

`main.cpp`:

```
#include <format>
#include <iostream>
#include <print>
#include <string>
#include <vector>

#include "calc.h"

static std::string com_virgula(double v) {
    std::string s = std::format("{:.15g}", v);  // 15 dígitos escondem o ruído do double
    for (char& c : s)
        if (c == '.')
            c = ',';
    return s;
}

int main() {
    calc::Calculadora c;
    std::vector<std::string> historico;
    std::string linha;
    std::println("Calculadora: + - * / ^ % e raiz. Comandos: hist, sair.");
    while (std::print("> "), std::getline(std::cin, linha)) {
        if (linha == "sair")
            break;
        if (linha == "hist") {
            for (std::size_t i = 0; i < historico.size(); ++i)
                std::println("  {}. {}", i + 1, historico[i]);
            continue;
        }
        if (linha.empty())
            continue;
        if (auto r = c.avaliar(linha)) {
            historico.push_back(linha + " = " + com_virgula(*r));
            std::println("{}", historico.back());
        } else {
            std::println("erro: {}", calc::mensagem(r.error()));
        }
    }
}
```

Entrada digitada:

```
2 + 3
0,1 + 0,2
ans * 10
10 / 0
2 ^ 0,5
raiz -4
raiz 2
15 % 200
abc + 1
2 ^ 2000
2 $ 3
hist
sair
```

Saída

```
Calculadora: + - * / ^ % e raiz. Comandos: hist, sair.
> 2 + 3 = 5
> 0,1 + 0,2 = 0,3
> ans * 10 = 3
> erro: divisão por zero
> erro: o expoente precisa ser inteiro
> erro: raiz de número negativo
> raiz 2 = 1,4142135623731
> 15 % 200 = 30
> erro: operando não é um número
> erro: resultado grande demais
> erro: operação desconhecida (use + - * / ^ % ou raiz)
>   1. 2 + 3 = 5
  2. 0,1 + 0,2 = 0,3
  3. ans * 10 = 3
  4. raiz 2 = 1,4142135623731
  5. 15 % 200 = 30
>
```

Compile os dois `.cpp` juntos: `clang++ -std=c++23 -Wall -Wextra calc.cpp main.cpp -o calc`. Com a entrada redirecionada, o terminal não ecoa o que foi digitado, e cada resposta aparece logo depois do `>`.

O `{:.15g}` está escondendo um erro de propósito: `0,1 + 0,2` guarda 0,30000000000000004, e `ans * 10` guarda 3,0000000000000004. Com `{}`, o `std::format` mostraria a representação mais curta que identifica o `double` exato, e o erro apareceria. Calculadoras de verdade usam aritmética decimal ou guardam mais dígitos do que mostram. Uma boa extensão é uma classe `Fracao` (como a do módulo 2) com `long long`, que torna exatas as quatro operações e a potência inteira.

Uma versão gráfica fica como extensão natural do módulo 10: com raylib, a janela desenha um visor e uma grade de botões, monta o texto da conta a cada clique e chama o mesmo `Calculadora::avaliar`, sem mudar uma linha do núcleo. É a prova de que separar o núcleo da interface valeu a pena.

D2

Um banco chave-valor persistente

Escreva um banco chave-valor de linha de comando com os comandos `set chave valor`, `get chave`, `del chave`, `list` e `sair`. Os dados devem sobreviver entre execuções: cada alteração é acrescentada a um arquivo de log (`set` e `del`), e ao iniciar o programa reconstrói o estado relendo o log. Acrescente um comando `compactar` que reescreve o log só com o estado atual. Teste o núcleo com Catch2.

Caminho da solução

- **Núcleo**: `class Banco` com um `std::unordered_map<std::string, std::string>` e os métodos `set`, `get` (devolve `std::optional<std::string>`) e `del` (devolve `bool`). Erros de uso, como um comando com argumentos faltando, voltam como `std::expected<…, std::string>` (o padrão do módulo 6).
- **Log**: uma classe `Log` com RAII sobre um `std::ofstream` aberto em modo `std::ios::app`; cada operação vira uma linha, como `set\tnome\tAna`. Use um separador que não pode aparecer nos dados, ou *escape* as tabulações e quebras de linha, senão um valor com `\t` corrompe a releitura.
- **Durabilidade**: após cada escrita, `flush()`. Para resistir a queda de energia, seria preciso também `fsync` no descritor: o `flush` só entrega os dados ao sistema operacional.
- **Compactação segura**: escreva o estado num arquivo temporário (`dados.log.tmp`), feche-o e só então troque com `std::filesystem::rename`, que substitui o original de uma vez. Se o programa cair no meio, o log antigo continua inteiro.
- **Recuperação**: ao reler o log, uma última linha incompleta (o programa morreu no meio da escrita) deve ser ignorada com um aviso, não derrubar o banco.
- **Testes**: casos do Catch2 para `set` seguido de `get`, `del` de chave inexistente, releitura de um log com uma linha corrompida e compactação seguida de releitura com o mesmo estado. Use um diretório temporário por teste (`std::filesystem::temp_directory_path()`), para que os testes não dependam uns dos outros.

É a ideia por trás de bancos como o Redis (com o arquivo AOF) e o Bitcask: um log só de acréscimos é simples, rápido para escrever e fácil de recuperar.

Fim do gabarito.
Dominando C++ em 1 Ano · theknowledgeisle.com