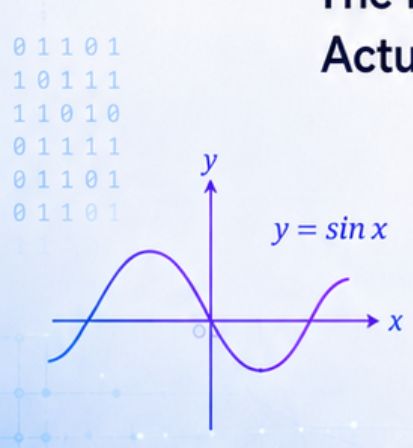
# No-Fear Math

## for

# C/C++ Software Engineers

**The Essential Mathematics C and C++ Engineers Actually Need — Explained Simply, Step by Step**




$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$$A = \begin{bmatrix} 1 & 2 & -1 \\ 0 & 1 & 3 \\ 4 & -2 & 1 \end{bmatrix}$$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$


$$P(A \cap B)$$

DRAFT

MASTERY

APPLICATION

MODELING

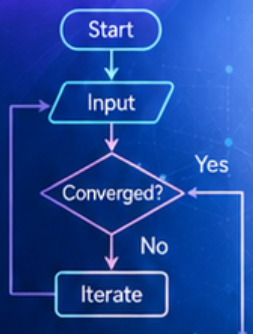
UNDERSTANDING

FOUNDATIONS

```
double dot(const std::vector<double>& a,
           const std::vector<double>& b) {
    std::size_t n = std::min(a.size(),b.size());
    double sum = 0.0;
    for (std::size_t i = 0; i < n; ++i) {
        sum += a[i] * b[i];
    }
    return sum;
}
```

```
#include <vector>
#include <cmath>
#include <cstddef>

std::vector<std::vector<double>> multiply(
    const std::vector<std::vector<double>>& A,
    const std::vector<std::vector<double>>& B) {
    std::size_t n = A.size();
    std::size_t m = B[0].size();
    std::size_t p = B.size();
    std::vector<std::vector<double>> C(n,
        std::vector<double>(m, 0.0));
    for (std::size_t i = 0; i < n; ++i) {
        for (std::size_t k = 0; k < p; ++k) {
            double aik = A[i][k];
            for (std::size_t j = 0; j < m; ++j) {
                C[i][j] += aik * B[k][j];
            }
        }
    }
    return C;
}
```




$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$


Carefully designed with the assistance of AI.

LINEAR ALGEBRA | CALCULUS | PROBABILITY & STATISTICS | DISCRETE MATH | ALGORITHMS & LOGIC

Prepared by **Ayman Alheraki**

# No-Fear Math for Software Engineers

## C & C++ Engineering Edition

The mathematics behind systems programming, algorithms, embedded software, graphics, games, HPC, networking, DSP, security, robotics, scientific computing, and more - explained simply, step by step.

**Purpose.** This book is not a university mathematics sequence compressed into a programmer's manual. It is a practical map of the mathematics a C or C++ software engineer repeatedly needs to understand code, design reliable algorithms, reason about hardware, and enter specialized domains without fearing the formulas.

Prepared by **Ayman Alheraki**

First C/C++ Engineering Edition - August 2026

# How to Use This Book

The chapters are arranged as a learning ramp. Early chapters build the arithmetic and discrete habits used everywhere. The middle of the book moves into machine arithmetic, performance, numerical reliability, vectors, and calculus. The final part shows how the same mathematics combines inside major C/C++ specialties.

**Priority scale**

**4-6: Useful foundation.** You should be comfortable with it.
**7-8: Important engineering math.** It appears frequently in algorithms and systems.
**9: Essential in several C/C++ specialties.** Weakness here causes real design limitations.
**10: Critical.** For the specialties that depend on it, this knowledge directly affects correctness, safety, or performance.

Every chapter is intentionally split into three layers: **concept map**, **C/C++ engineering lab**, and **reliability page**. Read the first page to understand the idea, the second to connect it to code, and the third to learn what usually fails in production.

## A note on completeness

No finite book can reproduce all of pure mathematics. This book instead aims for practical completeness: it covers the mathematical foundations and engineering patterns that repeatedly matter across the major areas in which C and C++ are used. Specialized research work can then go deeper from a strong map rather than from a blank page.

# C and C++ Numerical Ground Rules

- Unsigned integer arithmetic is defined modulo $2^N$ for an $N$-bit unsigned type. Signed overflow is undefined behavior in C and C++ and must not be used as an intentional wrap-around mechanism.
- A mathematically correct expression can still overflow, underflow, narrow, lose precision, or evaluate in an unexpected promoted type.
- Floating point is finite approximate arithmetic. Its results depend on representation, rounding, evaluation order, exceptional values, and compiler/hardware choices.
- Units, coordinate frames, time bases, scales, and probability assumptions are part of the type-level meaning even when the language represents them with ordinary numbers.
- Use the standard library where it expresses intent: `<cmath>`, `<numbers>`, `<complex>`, `<numeric>`, `<bit>`, `<random>`, `<chrono>`, `<limits>`, `<ratio>`, and related facilities. In C, know `math.h`, `stdint.h`, `limits.h`, `float.h`, `fenv.h`, and the exact requirements of the target environment.
- Prefer reviewed numerical, linear-algebra, signal-processing, and cryptographic libraries when the problem is bigger than a small educational implementation.

**Failure modes**

This book uses compact C++ examples to expose the mathematics. They are teaching fragments, not drop-in security, cryptography, hard-real-time, or safety-certified production libraries.

# Contents

Part I

# Practical Foundations

Useful mathematics that appears everywhere in C and C++ code. These chapters are deliberately gentle: the goal is to make quantities, scale, formulas, and machine representations feel routine before the more demanding topics begin.

CHAPTER 1

# Units, Dimensions, Ratios, and Scale

**Priority 4/10**

**Why this matters in C/C++**

Many software defects that look like logic bugs are actually unit bugs. C and C++ make it easy to mix bytes with elements, milliseconds with seconds, or radians with degrees unless the engineer keeps the mathematics explicit.

## Core ideas

- A quantity is a number plus a meaning and usually a unit.
- Ratios compare quantities of the same kind; rates compare different kinds.
- Dimensional analysis checks whether an equation can possibly be correct.
- Scale factors convert between representations such as bytes, KiB, MiB, and GiB.
- Percent change is relative; an absolute change is not the same thing.
- Normalize units at system boundaries so the core algorithm uses one convention.

**Core mathematical tools**

$$\text{rate} = \frac{\text{amount}}{\text{time}}$$

$$\text{percent change} = 100\frac{\text{new} - \text{old}}{\text{old}}$$

$$1\,\text{KiB} = 1024\,\text{B}$$

$$\text{scaled} = \text{value} \times \text{conversion factor}$$

**Where you will use it**

- memory sizing and allocator accounting
- network throughput and transfer time
- frame-time budgets
- sensor conversion in embedded systems
- file formats and storage capacity

CHAPTER 1

# Units, Dimensions, Ratios, and Scale - Engineering Lab

**Priority 4/10**

## Worked example

A buffer holds 48,000 stereo audio frames. Each channel uses a 32-bit float. The correct memory calculation is frames x channels x bytes-per-sample: 48,000 x 2 x 4 = 384,000 bytes. Writing the units beside each factor makes the cancellation visible and prevents counting samples as frames.

## C++ connection

```cpp
#include <cstddef>
#include <cstdint>

constexpr std::size_t bytes_for_audio(
   std::size_t frames, std::size_t channels) {
   constexpr std::size_t bytes_per_sample = sizeof(float);
   return frames * channels * bytes_per_sample;
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** memory sizing and allocator accounting, network throughput and transfer time, frame-time budgets, sensor conversion in embedded systems, file formats and storage capacity.

CHAPTER 1

# Units, Dimensions, Ratios, and Scale - Reliability Notes

**Priority 4/10**

**Failure modes**

- Mixing decimal MB with binary MiB.
- Converting units twice at two different layers.
- Integer division truncating a rate.
- Using 1000 where a protocol or format specifies 1024, or vice versa.
- Hiding units inside vague variable names such as t, d, or size.

**Engineering checks**

- Use names such as bytes, frames, seconds, and radians.
- Test known conversion identities.
- Use wider intermediates before multiplying large dimensions.
- Consider strong quantity types when unit mistakes would be expensive.

## Build intuition

A network link is 2.5 Gbit/s. Estimate the best-case time to transfer a 750 MiB file. Write every unit and conversion explicitly before computing.

**Chapter takeaway**

Before optimizing an equation, make sure its units are meaningful. Dimensional sanity is one of the cheapest correctness tools in engineering.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 2

# Algebra for Engineering Formulas

**Priority 5/10**

**Why this matters in C/C++**

Algebra is the language used to rearrange performance models, geometry equations, capacity formulas, and physical relationships. You rarely solve school-style equations for their own sake; you solve for the variable your program must compute.

## Core ideas

- An equation states that two expressions represent the same quantity.
- Apply the same reversible operation to both sides to isolate an unknown.
- Inequalities need special care when multiplying or dividing by a negative value.
- Factoring exposes common work and can improve both understanding and code.
- Piecewise formulas map naturally to branches or clamp operations.
- Symbolic simplification often reveals overflow-safe or numerically stable forms.

**Core mathematical tools**

$$ax + b = c \Rightarrow x = \frac{c - b}{a}$$

$$\frac{a}{b} = \frac{c}{d} \Rightarrow ad = bc$$

$$\min(\max(x, L), H) \in [L, H]$$

$$(a + b)^2 = a^2 + 2ab + b^2$$

**Where you will use it**

- interpolation and scaling
- capacity planning
- physics and simulation
- graphics coordinate conversion
- solving threshold conditions in algorithms

CHAPTER 2

# Algebra for Engineering Formulas - Engineering Lab

**Priority 5/10**

## Worked example

Suppose an image coordinate x in [0, 1919] must map to normalized device space [-1, 1]. Start from an affine form y = ax + b. Enforce y(0)=-1 and y(1919)=1. This gives b=-1 and a=2/1919. The derivation is short, but it is exactly the kind of algebra used constantly in graphics and UI code.

## C++ connection

```cpp
#include <algorithm>

double to_ndc(double x, double width) {
   if (width <= 1.0) return 0.0;
   const double y = 2.0 * x / (width - 1.0) - 1.0;
   return std::clamp(y, -1.0, 1.0);
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** interpolation and scaling, capacity planning, physics and simulation, graphics coordinate conversion, solving threshold conditions in algorithms.

CHAPTER 2

# Algebra for Engineering Formulas - Reliability Notes

**Priority 5/10**

**Failure modes**

- Rearranging a formula into a form that divides by a near-zero value.
- Forgetting integer-versus-floating arithmetic.
- Assuming multiplication cannot overflow before a later division reduces the value.
- Ignoring the valid domain of a square root or logarithm.
- Using a mathematically correct form that is numerically fragile.

**Engineering checks**

- Check boundary values and degenerate inputs.
- Use symbolic reasoning before coding.
- Prefer a form that keeps intermediates in range.
- Document the allowed domain of each variable.

## Build intuition

Derive an affine mapping that converts Celsius values in [-40, 125] into an unsigned 12-bit integer range [0, 4095]. Then determine the inverse mapping.

**Chapter takeaway**

Algebra lets you choose the safest computational form of the same mathematical relationship.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 3

# Functions, Graphs, Interpolation, and Clamping

**Priority 5/10**

**Why this matters in C/C++**

A software function often implements a mathematical function: input, transformation, output. Thinking in domains, ranges, monotonicity, and continuity makes APIs easier to reason about and edge cases easier to test.

## Core ideas

- The domain is the set of allowed inputs; the range is the set of possible outputs.
- A monotone function preserves ordering and is easier to search.
- Linear interpolation blends two endpoints with a parameter t.
- Clamping restricts a value to a legal interval.
- Inverse functions undo a one-to-one mapping.
- Piecewise curves are common in UI easing, color mapping, and control logic.

**Core mathematical tools**

$$\text{lerp}(a, b, t) = a + t(b - a)$$

$$\text{clamp}(x, L, H) = \min(\max(x, L), H)$$

$$y = mx + c$$

$f^{-1}(f(x)) = x$ when the inverse exists

**Where you will use it**

- animation and games
- graphics and color conversion
- sensor calibration
- binary search on monotone predicates
- table lookup and approximation

CHAPTER 3

# Functions, Graphs, Interpolation, and Clamping - Engineering Lab

**Priority 5/10**

## Worked example

A thermostat sensor reports 0.2 V at 0 C and 2.8 V at 100 C. If its response is approximately linear, a voltage of 1.5 V corresponds to 50 C. This is linear interpolation after normalizing the input into a 0-to-1 parameter.

## C++ connection

```cpp
#include <algorithm>
#include <cmath>

double map_range(double x, double in0, double in1,
                 double out0, double out1) {
   if (in0 == in1) return out0;
   double t = (x - in0) / (in1 - in0);
   t = std::clamp(t, 0.0, 1.0);
   return std::lerp(out0, out1, t);
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** animation and games, graphics and color conversion, sensor calibration, binary search on monotone predicates, table lookup and approximation.

CHAPTER 3

# Functions, Graphs, Interpolation, and Clamping - Reliability Notes

**Priority 5/10**

**Failure modes**

- Extrapolating unintentionally when t leaves [0,1].
- Using exact floating equality to detect a degenerate interval.
- Ignoring non-linear sensor response.
- Confusing interpolation of angles with interpolation of ordinary scalars.
- Clamping too early and hiding a real upstream error.

**Engineering checks**

- Test both endpoints exactly.
- Test a midpoint with a known expected result.
- Test reversed and degenerate intervals.
- Separate validation from intentional saturation.

## Build intuition

For an input range [10, 70] and output range [1000, 4000], compute the mapped result for x=25 and verify it using the normalized t value.

**Chapter takeaway**

Functions become easier to test when you can state their domain, range, monotonicity, and boundary behavior.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 4

# Powers, Roots, Exponentials, and Logarithms

**Priority 6/10**

**Why this matters in C/C++**

Powers and logarithms appear in complexity, bit widths, decibels, exponential backoff, hash table sizing, floating-point ranges, signal processing, and growth models.

## Core ideas

- A power multiplies a base by itself repeatedly; roots reverse powers.
- Logarithms answer: what exponent produces this value?
- Base 2 is natural for binary systems; base e is natural for continuous growth and calculus.
- Changing logarithm base changes only a constant factor.
- Exponential growth outruns every fixed-degree polynomial eventually.
- Log-space calculations can turn products into sums and avoid underflow.

**Core mathematical tools**

$$b^{x+y} = b^x b^y$$

$$\log_b(xy) = \log_b x + \log_b y$$

$$\log_b x = \frac{\ln x}{\ln b}$$

$\lceil \log_2 N \rceil$ bits can index roughly $N$ states

**Where you will use it**

- algorithm complexity
- binary trees and bit widths
- exponential backoff
- audio decibels
- probability likelihoods
- floating-point exponent reasoning

CHAPTER 4

# Powers, Roots, Exponentials, and Logarithms - Engineering Lab

**Priority 6/10**

## Worked example

If a table capacity is maintained as a power of two, a requested capacity of 1000 can be rounded to 1024. The corresponding exponent is 10 because 2^10=1024. C++20 provides bit operations that express this intent more directly than floating logarithms.

## C++ connection

```cpp
#include <bit>
#include <cstddef>

std::size_t next_pow2(std::size_t n) {
   if (n <= 1) return 1;
   return std::size_t{1} << std::bit_width(n - 1);
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** algorithm complexity, binary trees and bit widths, exponential backoff, audio decibels, probability likelihoods, floating-point exponent reasoning.

CHAPTER 4

# Powers, Roots, Exponentials, and Logarithms - Reliability Notes

**Priority 6/10**

**Failure modes**

- Using floating log2 to compute an exact integer bit count.
- Shifting by the width of the type or more.
- Overflowing when rounding a very large value to the next power of two.
- Confusing natural log with base-10 log.
- Applying a logarithm to zero or a negative input.

**Engineering checks**

- Use integer bit operations for integer capacity problems.
- Check maximum representable input before shifting.
- Test exact powers of two and values immediately around them.
- State the logarithm base when it matters to the interpretation.

## Build intuition

How many bits are needed to represent 1,000,000 distinct non-negative identifiers? Compare floor(log2 N), ceil(log2 N), and std::bit_width(N-1).

**Chapter takeaway**

Logs are not abstract decoration: they are the natural measuring tool for multiplicative growth and binary scale.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 5

# Binary, Hexadecimal, and Positional Number Systems

**Priority 6/10**

**Why this matters in C/C++**

C and C++ expose the machine closely. Binary and hexadecimal are therefore not optional notation: they are how masks, addresses, encodings, flags, SIMD lanes, and protocol fields are understood.

## Core ideas

- A positional numeral is a weighted sum of digits times powers of its base.
- Hexadecimal groups binary digits in blocks of four.
- Bit positions are zero-based powers of two.
- Masks select, clear, set, or toggle chosen bit positions.
- Endianness concerns byte order in memory, not the meaning of a numeric value in a register.
- Signed representation and integer conversion rules must be separated from bit-pattern reasoning.

**Core mathematical tools**

$$N = \sum_{i=0}^{k} d_i b^i$$

$1$ hex digit $= 4$ bits

mask for bit $k = 2^k$

$x$ & $(1 \ll k) \neq 0$ tests bit $k$

**Where you will use it**

- protocol headers
- device registers
- binary file formats
- allocators and alignment
- graphics packed pixels
- debugging machine-level state

CHAPTER 5

# Binary, Hexadecimal, and Positional Number Systems - Engineering Lab

**Priority 6/10**

## Worked example

The hexadecimal byte 0xB6 is binary 1011 0110. Bits 7, 5, 4, 2, and 1 are set. Being able to move between these views quickly makes register descriptions and packet layouts much easier to audit.

## C++ connection

```cpp
#include <cstdint>

bool has_flag(std::uint32_t value, unsigned bit) {
   if (bit >= 32) return false;
   return (value & (std::uint32_t{1} << bit)) != 0;
}

void set_flag(std::uint32_t& value, unsigned bit) {
   if (bit < 32) value |= (std::uint32_t{1} << bit);
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** protocol headers, device registers, binary file formats, allocators and alignment, graphics packed pixels, debugging machine-level state.

CHAPTER 5

# Binary, Hexadecimal, and Positional Number Systems - Reliability Notes

**Priority 6/10**

**Failure modes**

- Shifting a 32-bit literal before widening it.
- Assuming host byte order equals network byte order.
- Treating a signed integer as a portable bag of bits.
- Using magic hexadecimal constants without named masks.
- Ignoring padding or reserved bits in hardware fields.

**Engineering checks**

- Use unsigned fixed-width types for raw bit fields.
- Name masks and document bit positions.
- Test round-trip pack/unpack operations.
- Inspect boundary values such as bit 0 and the top legal bit.

## Build intuition

Write 0x9D3A in binary, identify the set bits, and compute the decimal value using positional weights rather than a calculator.

**Chapter takeaway**

Hex is a compact human view of binary structure. Learn to see fields and powers of two, not just digits.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 6

# Boolean Algebra and Bitwise Logic

**Priority 6/10**

**Why this matters in C/C++**

Branch conditions, feature flags, state machines, masks, predicates, compiler optimizations, and digital interfaces are all built on Boolean logic. The same algebra explains both logical operators and many bitwise transformations.

## Core ideas

- AND is true only when both inputs are true; OR is true when at least one is true.
- XOR is true when inputs differ.
- NOT complements a Boolean value or every bit in a mask.
- De Morgan laws transform negated conjunctions and disjunctions.
- Short-circuit logical operators are not the same as bitwise operators.
- Truth tables are a mechanical way to validate small logical expressions.

**Core mathematical tools**

$$\neg(A \wedge B) = \neg A \vee \neg B$$

$$\neg(A \vee B) = \neg A \wedge \neg B$$

$$A \oplus A = 0$$

$$A \oplus 0 = A$$

**Where you will use it**

- permission checks
- state machines
- bit masks
- compiler conditions
- embedded register manipulation
- SIMD predicate masks

CHAPTER 6

# Boolean Algebra and Bitwise Logic - Engineering Lab

**Priority 6/10**

## Worked example

Suppose access is allowed when a user is authenticated and either owns the resource or has admin privilege. The expression auth && (owner || admin) is clearer and safer than distributing the condition across several branches. A truth table can verify all eight input combinations.

## C++ connection

```
bool can_write(bool authenticated, bool owner, bool admin) {
   return authenticated && (owner || admin);
}

std::uint32_t toggle_mask(std::uint32_t value,
                  std::uint32_t mask) {
   return value ^ mask;
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** permission checks, state machines, bit masks, compiler conditions, embedded register manipulation, SIMD predicate masks.

CHAPTER 6

# Boolean Algebra and Bitwise Logic - Reliability Notes

**Priority 6/10**

**Failure modes**

- Using & when && is required, causing both operands to execute.
- Negating a complex condition incorrectly.
- Relying on operator precedence for mixed logical and bitwise expressions.
- Packing unrelated state into opaque masks.
- Forgetting that XOR is not exponentiation.

**Engineering checks**

- Parenthesize mixed logical expressions for readability.
- Build truth-table tests for security-sensitive predicates.
- Separate pure predicates from expressions with side effects.
- Use enum-based flags rather than anonymous bit numbers where possible.

## Build intuition

Simplify !(ready && (cached || local)) using De Morgan laws. Then write a truth table to confirm that the transformed expression is equivalent.

**Chapter takeaway**

Boolean algebra is a correctness tool: it lets you prove that a condition says exactly what the code intends.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 7

# Sets, Relations, Maps, and Equivalence

**Priority 6/10**

**Why this matters in C/C++**

Sets and relations give precise language for containers, unique keys, permissions, graph edges, compiler analyses, cache membership, and partitioning. They also explain why some algorithms need equality and others need a strict ordering.

## Core ideas

- A set contains unique elements; order is not part of the mathematical set.
- Union combines memberships; intersection keeps common members; difference removes members.
- A relation is a set of ordered pairs.
- An equivalence relation is reflexive, symmetric, and transitive and partitions values into classes.
- A strict weak ordering is the mathematical contract behind ordered associative containers.
- A map associates keys with values; injective and bijective mappings matter for IDs and encodings.

**Core mathematical tools**

$$A \cup B$$

$$A \cap B$$

$$A \setminus B$$

$a \sim b \Rightarrow$ same equivalence class

**Where you will use it**

- std::set and std::unordered_set
- permissions and feature sets
- compiler data-flow sets
- deduplication
- graph connectivity
- canonicalization and interning

CHAPTER 7

# Sets, Relations, Maps, and Equivalence - Engineering Lab

**Priority 6/10**

## Worked example

If two file paths are considered equivalent after normalization, the normalization rule defines an equivalence relation only if it behaves consistently. If equivalence is inconsistent with hashing or ordering, C++ containers can silently misbehave.

## C++ connection

```cpp
#include <set>
#include <algorithm>

template<class T>
std::set<T> intersection(const std::set<T>& a,
                         const std::set<T>& b) {
   std::set<T> out;
   std::set_intersection(a.begin(), a.end(), b.begin(), b.end(),
                         std::inserter(out, out.end()));
   return out;
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** std::set and std::unordered_set, permissions and feature sets, compiler data-flow sets, deduplication, graph connectivity, canonicalization and interning.

CHAPTER 7

# Sets, Relations, Maps, and Equivalence - Reliability Notes

**Priority 6/10**

**Failure modes**

- Providing a comparator that is not a strict weak ordering.
- Using approximate floating equality as a map ordering relation.
- Confusing mathematical set membership with container iteration order.
- Hashing fields that equality ignores, or vice versa.
- Treating a partial relation as if it were a total order.

**Engineering checks**

- Test comparator irreflexivity and transitivity.
- Ensure equal keys produce equal hashes.
- Define canonical representations at boundaries.
- Use sets to express data-flow facts explicitly instead of ad-hoc flags.

## Build intuition

Give an example of a relation that is symmetric but not transitive. Then explain why such a relation is unsuitable as an equivalence rule for unordered_map keys.

**Chapter takeaway**

Container contracts are mathematical contracts. Equality, hashing, and ordering must agree with the relation you intend.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 8

# Sequences, Summations, and Series

**Priority 6/10**

**Why this matters in C/C++**

Loops are sequences. Running totals are summations. Memory offsets, triangular work, moving averages, and convergence criteria often become obvious when expressed with sigma notation.

## Core ideas

- A sequence is an ordered list indexed by position.
- A finite sum compresses repeated addition into one expression.
- Arithmetic sequences grow by a constant difference.
- Geometric sequences grow by a constant ratio.
- Prefix sums convert repeated range-sum queries into differences of cumulative totals.
- Convergent series provide the mathematical basis of many approximations.

**Core mathematical tools**

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=0}^{n-1} ar^i = a\frac{1-r^n}{1-r} \ (r \neq 1)$$

$$P_k = \sum_{i=0}^{k-1} x_i$$

$$\sum_{i=l}^{r-1} x_i = P_r - P_l$$

**Where you will use it**

- loop cost analysis
- prefix sums and integral images
- financial accumulation
- filtering and signal windows
- Taylor approximations

CHAPTER 8

# Sequences, Summations, and Series - Engineering Lab

**Priority 6/10**

## Worked example

A nested loop that performs 1 + 2 + ... + n operations does n(n+1)/2 work, which grows quadratically. This lets you predict scaling before benchmarking the implementation.

## C++ connection

```cpp
#include <vector>

std::vector<long long> prefix_sum(const std::vector<int>& x) {
   std::vector<long long> p(x.size() + 1, 0);
   for (std::size_t i = 0; i < x.size(); ++i)
      p[i + 1] = p[i] + x[i];
   return p;
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** loop cost analysis, prefix sums and integral images, financial accumulation, filtering and signal windows, Taylor approximations.

CHAPTER 8

# Sequences, Summations, and Series - Reliability Notes

**Priority 6/10**

**Failure modes**

- Summing into a type too narrow for the total.
- Off-by-one errors in half-open ranges.
- Applying a closed-form formula without checking overflow.
- Ignoring floating accumulation error.
- Assuming a series converges because its terms become small.

**Engineering checks**

- Use wider accumulators.
- Write index ranges explicitly as [begin,end).
- Cross-check a closed form against a small loop.
- Use compensated summation when error matters.

## Build intuition

A program executes i operations on iteration i for i=1..100000. Use the closed-form triangular sum and estimate whether a 32-bit signed counter can hold the total.

**Chapter takeaway**

Summation notation is the bridge between loop structure and mathematical reasoning about cost, storage, and error.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

Part II

# Discrete Mathematics and Algorithms

This is the mathematical backbone of data structures, algorithms, compilers, protocols, and systems code. The emphasis is not formalism for its own sake: every idea is tied to an engineering decision or correctness argument.

Progression: practical foundations → rigorous reasoning → machine reliability → specialist mathematics

CHAPTER 9

# Counting, Permutations, and Combinations

**Priority 7/10**

**Why this matters in C/C++**

Counting tells you how large a search space can become, how many test combinations exist, how many states a protocol has, and why brute force becomes impossible long before the code looks large.

## Core ideas

- The product rule multiplies independent choices.
- Permutations count ordered selections; combinations count unordered selections.
- Factorials grow extremely quickly.
- The pigeonhole principle proves that collisions or repetition must occur when there are more objects than slots.
- Inclusion-exclusion corrects double counting when sets overlap.
- Counting is often the first step in estimating probability.

**Core mathematical tools**

$$n! = n(n-1)\cdots 1$$

$$P(n,r) = \frac{n!}{(n-r)!}$$

$$\binom{n}{r} = \frac{n!}{r!(n-r)!}$$

$$|A \cup B| = |A| + |B| - |A \cap B|$$

**Where you will use it**

- test-case explosion
- password/key spaces
- state-machine analysis
- combinatorial search
- hash collision reasoning

CHAPTER 9

# Counting, Permutations, and Combinations - Engineering Lab

**Priority 7/10**

## Worked example

A feature system has 12 independent Boolean flags. It has 2^12 = 4096 possible configurations before any parameters are considered. This simple count explains why exhaustive testing can be cheap for 10 flags and unrealistic for 30.

## C++ connection

```cpp
#include <cstdint>
#include <optional>

std::optional<std::uint64_t> boolean_states(unsigned flags) {
   if (flags >= 64) return std::nullopt;
   return std::uint64_t{1} << flags;
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** test-case explosion, password/key spaces, state-machine analysis, combinatorial search, hash collision reasoning.

CHAPTER 9

# Counting, Permutations, and Combinations - Reliability Notes

**Priority 7/10**

**Failure modes**

- Computing factorials directly when only a ratio is needed.
- Forgetting whether order matters.
- Using floating point for exact combinatorial counts.
- Ignoring constraints that eliminate states.
- Underestimating exponential growth.

**Engineering checks**

- Model the choices before writing a brute-force loop.
- Use multiplicative formulas that reduce overflow risk.
- Compare search-space size with the actual operation budget.
- Use the pigeonhole principle to reason about unavoidable collisions.

## Build intuition

A test generator chooses 3 distinct codecs from 10, and order does not matter. How many combinations exist? How many if order determines a three-stage pipeline?

**Chapter takeaway**

Counting converts a vague “large search space” into a number that guides algorithm and test strategy.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 10

# Proof Thinking, Invariants, and Induction

**Priority 7/10**

**Why this matters in C/C++**

Production C/C++ code does not need to look like a mathematics paper, but strong engineers reason with proofs constantly: loop invariants, ownership invariants, state invariants, and inductive arguments about recursive structures.

## Core ideas

- An invariant is a statement that remains true at a chosen program point.
- A proof by induction has a base case and an inductive step.
- A loop invariant connects initialization, preservation, and termination.
- Proof by contradiction is useful for impossibility and uniqueness arguments.
- A counterexample disproves a universal claim.
- Assertions can encode selected invariants at runtime or during tests.

**Core mathematical tools**

$$P(0) \ \wedge \ \forall k \, (P(k) \Rightarrow P(k+1)) \Rightarrow \forall n \, P(n)$$

$$\text{initialization} + \text{preservation} + \text{termination} \Rightarrow \text{loop correctness}$$

**Where you will use it**

- binary search correctness
- container invariants
- memory ownership
- lock-free data structures
- compiler transforms
- recursive algorithms

CHAPTER 10

# Proof Thinking, Invariants, and Induction - Engineering Lab

**Priority 7/10**

## Worked example

For a partitioning loop, an invariant might state: every element before index i satisfies the predicate and every element from i to j is unclassified. Writing that sentence before coding often prevents subtle boundary errors.

## C++ connection

```cpp
#include <cassert>
#include <vector>

bool sorted_non_decreasing(const std::vector<int>& a) {
   for (std::size_t i = 1; i < a.size(); ++i) {
      assert(i <= a.size());
      if (a[i] < a[i - 1]) return false;
   }
   return true;
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** binary search correctness, container invariants, memory ownership, lock-free data structures, compiler transforms, recursive algorithms.

CHAPTER 10

# Proof Thinking, Invariants, and Induction - Reliability Notes

**Priority 7/10**

**Failure modes**

- Treating tests as proof for all possible inputs.
- Using an invariant that is too weak to imply correctness.
- Forgetting overflow or aliasing assumptions in a proof.
- Reasoning about a recursive case without proving termination.
- Writing assertions that themselves have side effects.

**Engineering checks**

- State the invariant in plain English.
- Check it before and after each state-changing operation in debug tests.
- Include representation and lifetime assumptions.
- Search for a minimal counterexample when a claim feels suspicious.

## Build intuition

Write a loop invariant for finding the maximum element in a non-empty array. Identify the base case, preservation step, and what the invariant says when the loop terminates.

**Chapter takeaway**

Proof thinking is disciplined debugging done before the bug exists.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 11

# Recurrences and Recursive Cost

**Priority 7/10**

**Why this matters in C/C++**

Divide-and-conquer algorithms, tree traversal, dynamic programming, parsers, and recursive data structures are naturally described by recurrence relations. Solving or estimating the recurrence predicts runtime and stack depth.

## Core ideas

- A recurrence defines a quantity using smaller instances of the same quantity.
- Linear recurrences often describe one-step dynamic programs.
- Divide-and-conquer recurrences combine subproblem cost with per-level work.
- Recursion depth can matter as much as total operation count in C/C++ because stack space is finite.
- Memoization changes repeated recursive expansion into stored subproblems.
- The Master Theorem is a useful pattern, not a substitute for understanding the recursion tree.

**Core mathematical tools**

$$T(n) = T(n-1) + O(1) \Rightarrow O(n)$$

$$T(n) = 2T(n/2) + O(n) \Rightarrow O(n \log n)$$

$$F_n = F_{n-1} + F_{n-2}$$

**Where you will use it**

- merge sort
- tree algorithms
- dynamic programming
- recursive-descent parsing
- divide-and-conquer geometry

CHAPTER 11

# Recurrences and Recursive Cost - Engineering Lab

**Priority 7/10**

## Worked example

Merge sort splits into two half-size problems and performs linear work to merge them. Every recursion level therefore costs about n work, and there are about log2(n) levels, giving n log n total work.

## C++ connection

```cpp
#include <span>

std::size_t recursion_levels(std::size_t n) {
   std::size_t levels = 0;
   while (n > 1) {
      n = (n + 1) / 2;
      ++levels;
   }
   return levels;
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** merge sort, tree algorithms, dynamic programming, recursive-descent parsing, divide-and-conquer geometry.

CHAPTER 11

# Recurrences and Recursive Cost - Reliability Notes

**Priority 7/10**

**Failure modes**

- Assuming recursive syntax automatically means exponential cost.
- Ignoring repeated subproblems.
- Forgetting stack depth and frame size.
- Applying the Master Theorem to a recurrence outside its assumptions.
- Dropping a non-constant term that actually dominates.

**Engineering checks**

- Draw the first three recursion levels.
- Count work per level and number of levels separately.
- Measure maximum stack depth.
- Compare recursive and iterative forms when depth can be attacker-controlled.

## Build intuition

For $T(n)=3T(n/2)+n$, sketch the recursion tree and determine whether work grows faster or slower than $n \log n$.

**Chapter takeaway**

A recurrence is a performance blueprint for recursive code.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 12

# Asymptotic Complexity: O, Theta, Omega, and Amortization

**Priority 8/10**

**Why this matters in C/C++**

C/C++ often operates where scale matters. Complexity notation separates growth behavior from machine-specific constants, while amortized analysis explains why occasional expensive operations can still produce excellent average cost per operation.

## Core ideas

- Big-O is an upper growth bound; Omega is a lower bound; Theta is a tight asymptotic bound.
- Constants and lower-order terms vanish asymptotically but still matter at realistic sizes.
- Worst-case, average-case, expected, and amortized costs are different claims.
- Amortized analysis spreads rare expensive events over many cheap operations.
- Space complexity includes auxiliary memory and sometimes recursion stack.
- Cache behavior can make two algorithms with the same Big-O perform very differently.

**Core mathematical tools**

$$f(n) \in O(g(n)) \iff \exists c, n_0 : \ f(n) \leq cg(n) \ \forall n \geq n_0$$

$$1 < \log n < n < n \log n < n^2 < 2^n < n!$$

**Where you will use it**

- container selection
- algorithm review
- API complexity contracts
- capacity growth policies
- performance regressions

CHAPTER 12

# Asymptotic Complexity: O, Theta, Omega, and Amortization - Engineering Lab

**Priority 8/10**

## Worked example

A vector that doubles capacity sometimes copies O(n) elements, yet repeated push_back operations are amortized O(1): across a long sequence, each existing element participates in only a small number of reallocations.

## C++ connection

```cpp
#include <vector>

void reserve_if_known(std::vector<int>& v, std::size_t expected) {
   // Complexity math tells us when avoiding geometric growth is useful.
   v.reserve(expected);
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** container selection, algorithm review, API complexity contracts, capacity growth policies, performance regressions.

CHAPTER 12

# Asymptotic Complexity: O, Theta, Omega, and Amortization - Reliability Notes

**Priority 8/10**

**Failure modes**

- Calling an operation O(1) without stating whether it is worst-case or amortized.
- Ignoring expensive hashing or comparison functions.
- Believing lower Big-O always wins at small n.
- Ignoring memory traffic and branch predictability.
- Using average-case claims for adversarial inputs.

**Engineering checks**

- Write the complexity contract next to performance-sensitive APIs.
- Benchmark across several input scales, not one size.
- Plot time/n, time/(n log n), or another normalized quantity.
- Consider memory and cache complexity alongside operation count.

## Build intuition

Compare n^2 with 1000n log2 n. Estimate the n where the asymptotically better algorithm begins to win under this deliberately large constant factor.

**Chapter takeaway**

Asymptotics tell you how software ages as input grows; measurements tell you where the crossover occurs today.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 13

# Graphs: Connectivity, Paths, and Dependency Structure

**Priority 8/10**

**Why this matters in C/C++**

Graphs model networks, build dependencies, call graphs, ownership relationships, game maps, compiler control flow, package dependencies, and state transitions. Once you see graph structure, many problems become standard traversals instead of custom logic.

## Core ideas

- A graph has vertices and edges; edges can be directed or undirected.
- Degree counts incident edges; in directed graphs distinguish in-degree and out-degree.
- A path is a sequence of connected vertices.
- BFS finds shortest paths in unweighted graphs; DFS is natural for reachability and structural analysis.
- A directed acyclic graph supports topological ordering.
- Weighted shortest-path algorithms depend on assumptions about edge weights.

**Core mathematical tools**

$$G = (V, E)$$

$$\text{BFS/DFS time} = O(|V| + |E|)$$

$$\sum_{v \in V} \deg(v) = 2|E| \text{ for undirected graphs}$$

**Where you will use it**

- dependency resolution
- routing
- compiler CFGs
- garbage collection
- build systems
- scene graphs and game maps

CHAPTER 13

# Graphs: Connectivity, Paths, and Dependency Structure - Engineering Lab

**Priority 8/10**

## Worked example

A build graph has a directed edge A -> B when B depends on A. A topological ordering gives a legal build order. A cycle means no such order exists and should be reported as a dependency error.

## C++ connection

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

std::vector<int> bfs(const std::vector<std::vector<int>>& g, int s) {
   std::vector<int> dist(g.size(), -1);
   std::queue<int> q; dist[s] = 0; q.push(s);
   while (!q.empty()) {
      int u = q.front(); q.pop();
      for (int v : g[u]) if (dist[v] < 0) {
         dist[v] = dist[u] + 1; q.push(v);
      }
   }
   return dist;
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** dependency resolution, routing, compiler CFGs, garbage collection, build systems, scene graphs and game maps.

CHAPTER 13

# Graphs: Connectivity, Paths, and Dependency Structure - Reliability Notes

**Priority 8/10**

**Failure modes**

- Using Dijkstra with negative edge weights.
- Representing a sparse graph with an unnecessarily huge dense matrix.
- Forgetting visited state and looping forever.
- Confusing tree edges with arbitrary graph edges.
- Recursing deeply with DFS on attacker-controlled graphs.

**Engineering checks**

- Choose adjacency lists or matrices based on density.
- Test disconnected graphs and single-vertex graphs.
- Detect cycles explicitly when acyclicity is a requirement.
- Use iterative traversal when recursion depth is unsafe.

## Build intuition

A graph has 1,000,000 vertices but only 3,000,000 edges. Explain why an adjacency matrix is usually a poor representation and estimate its bit-level storage before overhead.

**Chapter takeaway**

Graphs turn dependency-shaped problems into a small set of well-understood mathematical structures.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 14

# Trees, Heaps, Tries, and Hierarchical Cost

**Priority 8/10**

**Why this matters in C/C++**

Trees are the structure behind search trees, heaps, file systems, parsers, scene hierarchies, indexes, tries, and many allocator designs. Their height often determines latency.

## Core ideas

- A tree is a connected acyclic graph.
- Depth measures distance from the root; height measures the longest downward path.
- A balanced binary tree has logarithmic height.
- A heap gives fast access to an extremal element but is not globally sorted.
- A trie uses key symbols as edges and makes lookup cost depend on key length.
- B-trees increase branching factor to reduce height and I/O.

**Core mathematical tools**

$$\text{balanced binary height} \approx \lceil \log_2(n+1) \rceil - 1$$

$$\text{heap parent}(i) = \lfloor (i-1)/2 \rfloor$$

$$\text{left}(i) = 2i + 1$$

**Where you will use it**

- priority queues
- database indexes
- parsers and ASTs
- file systems
- routing tables
- spatial hierarchies

CHAPTER 14

# Trees, Heaps, Tries, and Hierarchical Cost - Engineering Lab

**Priority 8/10**

## Worked example

A binary heap stored in a vector needs no pointers. For node i, children are at 2i+1 and 2i+2. The mathematics turns a tree relation into simple index arithmetic, improving locality.

## C++ connection

```cpp
#include <cstddef>

constexpr std::size_t parent(std::size_t i) { return (i - 1) / 2; }
constexpr std::size_t left(std::size_t i) { return 2 * i + 1; }
constexpr std::size_t right(std::size_t i) { return 2 * i + 2; }
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** priority queues, database indexes, parsers and ASTs, file systems, routing tables, spatial hierarchies.

CHAPTER 14

# Trees, Heaps, Tries, and Hierarchical Cost - Reliability Notes

**Priority 8/10**

**Failure modes**

- Calling parent(0) with an unsigned index.
- Assuming a binary search tree stays balanced automatically.
- Confusing heap order with sorted order.
- Overflowing 2*i+2 on extreme indices.
- Ignoring pointer overhead and locality in node-heavy trees.

**Engineering checks**

- Check index arithmetic before dereference.
- Measure actual tree height in performance tests.
- Use iterative traversal for deep trees.
- Choose branching factor with the storage hierarchy in mind.

## Build intuition

For one million elements, estimate the ideal balanced binary-tree height and compare it with a B-tree node that can hold 128 children.

**Chapter takeaway**

Tree height translates structure into latency. Branching factor and balance determine how many decisions the program must make.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 15

# Modular Arithmetic and Wrap-Around Reasoning

**Priority 8/10**

**Why this matters in C/C++**

Modulo arithmetic appears in circular buffers, timers, hash tables, sequence numbers, checksums, alignment, periodic scheduling, and unsigned integer behavior.

## Core ideas

- a mod m is the remainder class of a relative to m.
- Congruence means two integers have the same remainder modulo m.
- Addition and multiplication can be performed modulo m.
- Powers of two make modulo operations compatible with bit masks for non-negative unsigned values.
- Unsigned C/C++ arithmetic is defined modulo 2^N for an N-bit unsigned type.
- Ordering wrapped counters requires interval assumptions; ordinary integer comparison may be wrong across wrap.

**Core mathematical tools**

$$a \equiv b \pmod{m} \iff m \mid (a-b)$$

$$(a+b) \bmod m = ((a \bmod m) + (b \bmod m)) \bmod m$$

$$x \bmod 2^k = x \;\&\; (2^k - 1) \text{ for unsigned } x$$

**Where you will use it**

- ring buffers
- sequence numbers
- periodic timers
- power-of-two hash tables
- alignment calculations

CHAPTER 15

# Modular Arithmetic and Wrap-Around Reasoning - Engineering Lab

**Priority 8/10**

## Worked example

For a circular buffer with capacity 1024, advancing index i can use (i+1) & 1023 when i is unsigned and capacity is exactly a power of two. The mask is not a magic trick; it is modulo 2^10.

## C++ connection

```cpp
#include <cstddef>

std::size_t next_ring(std::size_t i, std::size_t capacity) {
   return (i + 1) % capacity;
}

std::size_t next_pow2_ring(std::size_t i, std::size_t capacity) {
   return (i + 1) & (capacity - 1); // require power of two
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** ring buffers, sequence numbers, periodic timers, power-of-two hash tables, alignment calculations.

CHAPTER 15

# Modular Arithmetic and Wrap-Around Reasoning - Reliability Notes

**Priority 8/10**

**Failure modes**

- Using a mask when capacity is not a power of two.
- Applying signed % behavior without considering negative operands.
- Treating wrapped counters as ordinary ordered integers.
- Using signed overflow for intended wrap-around.
- Forgetting capacity==0 before modulo.

**Engineering checks**

- State the modulus explicitly in protocol and ring-buffer code.
- Use unsigned arithmetic for deliberate modulo 2^N behavior.
- Test immediately before and after wrap.
- Prove power-of-two preconditions if replacing % with &.

## Build intuition

A 16-bit sequence number wraps after 65535. If messages can be at most 1000 positions apart, design a rule for deciding whether one sequence number is newer than another across wrap.

**Chapter takeaway**

Modulo arithmetic is the correct model for cyclic state; treating cycles as a straight number line creates bugs.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 16

# GCD, Primes, Finite Fields, and Number-Theory Tools

**Priority 8/10**

**Why this matters in C/C++**

Number theory becomes practical in cryptography, hashing, rational timing, periodic schedules, pseudo-random generators, CRCs, and modular inverses. C/C++ engineers do not need every theorem, but they do need the core tools.

## Core ideas

- The greatest common divisor measures the largest shared factor.
- Euclid’s algorithm computes gcd efficiently.
- The least common multiple helps combine periods.
- A prime has no positive divisors except 1 and itself.
- A modular inverse of a exists modulo m exactly when gcd(a,m)=1.
- Finite fields support division by every non-zero element and are fundamental in coding and cryptography.

**Core mathematical tools**

$$\gcd(a,b) = \gcd(b, a \bmod b)$$

$$\operatorname{lcm}(a,b) = \frac{|ab|}{\gcd(a,b)}$$

$$aa^{-1} \equiv 1 \pmod{m}$$

**Where you will use it**

- periodic task alignment
- cryptography
- CRC and error detection
- hashing
- rational sample-rate conversion

CHAPTER 16

# GCD, Primes, Finite Fields, and Number-Theory Tools - Engineering Lab

**Priority 8/10**

## Worked example

Two periodic tasks run every 12 ms and 18 ms. Their pattern repeats every lcm(12,18)=36 ms. This simple number-theory result gives the hyperperiod used in schedule analysis.

## C++ connection

```cpp
#include <numeric>
#include <cstdint>

std::int64_t hyperperiod(std::int64_t a, std::int64_t b) {
   return std::lcm(a, b);
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** periodic task alignment, cryptography, CRC and error detection, hashing, rational sample-rate conversion.

CHAPTER 16

# GCD, Primes, Finite Fields, and Number-Theory Tools - Reliability Notes

**Priority 8/10**

**Failure modes**

- Multiplying a*b before gcd reduction and overflowing lcm calculations.
- Assuming every non-zero residue has an inverse modulo a composite modulus.
- Using textbook modular exponentiation as production cryptography.
- Treating primality testing as trial division for huge cryptographic integers.
- Mixing GF(2) polynomial arithmetic with ordinary integer arithmetic.

**Engineering checks**

- Reduce before multiplying when possible.
- Check gcd(a,m)==1 before asking for an inverse.
- Use reviewed cryptographic libraries for security.
- Separate integer modulus from polynomial-field notation.

## Build intuition

Compute gcd(252,105) with Euclid’s algorithm, then compute the lcm without first multiplying 252*105.

**Chapter takeaway**

A small number-theory toolkit unlocks many cyclic, cryptographic, scheduling, and coding problems.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

Part III

# Probability, Statistics, and Uncertainty

Software meets uncertainty in workloads, latency, failures, sampling, randomized algorithms, hashing, telemetry, experiments, machine learning, and simulation. The goal here is to reason correctly about variation instead of hiding it behind one average.

Progression: practical foundations → rigorous reasoning → machine reliability → specialist mathematics

CHAPTER 17

# Probability Rules, Conditional Probability, and Bayes

**Priority 7/10**

**Why this matters in C/C++**

Probability gives a numerical language for uncertainty. C/C++ engineers use it when analyzing reliability, caches, tests, randomized algorithms, sensors, network loss, and evidence-driven decisions.

## Core ideas

- A probability lies between 0 and 1.
- The complement rule often simplifies “at least one” questions.
- Conditional probability changes the sample space after evidence is known.
- Independence means learning one event does not change the probability of another.
- Bayes theorem reverses a conditional probability using prior information.
- Union probabilities require subtracting overlap to avoid double counting.

**Core mathematical tools**

$$P(A^c) = 1 - P(A)$$

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)}$$

$$P(A \cup B) = P(A) + P(B) - P(A \cap B)$$

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}$$

**Where you will use it**

- fault detection
- randomized tests
- network loss models
- sensor fusion
- security alerts
- cache-hit reasoning

CHAPTER 17

# Probability Rules, Conditional Probability, and Bayes - Engineering Lab

**Priority 7/10**

## Worked example

If an independent operation fails with probability 0.001, the probability that at least one of 100 operations fails is 1-(0.999)^100, about 9.5 percent. Multiplying 0.001 by 100 gives a useful small-probability approximation, but the complement calculation is the exact independent model.

## C++ connection

```cpp
#include <cmath>

double at_least_one_failure(double p, std::size_t n) {
   if (p <= 0.0) return 0.0;
   if (p >= 1.0) return n ? 1.0 : 0.0;
   return -std::expm1(static_cast<double>(n) * std::log1p(-p));
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** fault detection, randomized tests, network loss models, sensor fusion, security alerts, cache-hit reasoning.

CHAPTER 17

# Probability Rules, Conditional Probability, and Bayes - Reliability Notes

**Priority 7/10**

**Failure modes**

- Adding probabilities for events that overlap.
- Assuming independence because events look unrelated.
- Confusing P(A|B) with P(B|A).
- Using a rare-event approximation outside its useful range.
- Treating a model probability as a guaranteed long-run frequency.

**Engineering checks**

- Write the sample space and assumptions.
- Use complements for “at least one” events.
- Validate independence assumptions with system knowledge.
- Test probability code at p=0, p=1, and very small p.

## Build intuition

A detector catches 99 percent of real faults but produces a false alarm on 1 percent of healthy runs. If only 0.1 percent of runs actually contain a fault, use Bayes reasoning to explain why a positive alarm is not 99 percent certain to indicate a real fault.

**Chapter takeaway**

Probability is only as good as its assumptions. State independence, priors, and event definitions before trusting the number.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 18

# Random Variables, Expectation, Variance, and Covariance

**Priority 7/10**

**Why this matters in C/C++**

A random variable converts uncertain outcomes into numbers. Expectation predicts long-run average cost; variance measures spread; covariance reveals whether quantities move together.

## Core ideas

- Expected value is a probability-weighted average.
- Variance measures squared distance from the mean.
- Standard deviation has the same unit as the original variable.
- Linearity of expectation does not require independence.
- Covariance measures joint variation; correlation normalizes it.
- The expectation of a non-linear function is generally not the function of the expectation.

**Core mathematical tools**

$$E[X] = \sum_x xP(X = x)$$

$$\text{Var}(X) = E[(X - E[X])^2]$$

$$\text{Var}(X) = E[X^2] - E[X]^2$$

$$\text{Cov}(X, Y) = E[(X - E[X])(Y - E[Y])]$$

**Where you will use it**

- expected algorithm cost
- latency variation
- simulation
- risk models
- performance counters
- sensor noise

CHAPTER 18

# Random Variables, Expectation, Variance, and Covariance - Engineering Lab

**Priority 7/10**

## Worked example

A cache access costs 4 ns on a hit and 80 ns on a miss. With a 95 percent hit rate, expected access time is 0.95*4 + 0.05*80 = 7.8 ns. This expectation is useful for throughput modeling, but it does not tell you the tail latency of a dependency chain.

## C++ connection

```cpp
#include <span>

double mean(std::span<const double> x) {
   double s = 0.0;
   for (double v : x) s += v;
   return x.empty() ? 0.0 : s / x.size();
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** expected algorithm cost, latency variation, simulation, risk models, performance counters, sensor noise.

CHAPTER 18

# Random Variables, Expectation, Variance, and Covariance - Reliability Notes

**Priority 7/10**

**Failure modes**

- Using only a mean for a heavy-tailed latency distribution.
- Computing variance with an unstable one-pass formula for extreme values.
- Assuming zero covariance means full statistical independence.
- Averaging rates incorrectly.
- Forgetting weighted probabilities.

**Engineering checks**

- Report a spread measure with the mean.
- Use numerically stable online algorithms for large streams.
- Plot or inspect the distribution before selecting summaries.
- Separate independent and correlated sources of variation.

## Build intuition

A branch costs 1 unit when predicted and 15 units when mispredicted. Derive expected cost as a function of misprediction probability p. Evaluate it for p=0.02 and p=0.20.

**Chapter takeaway**

Expectation answers “what on average?”; variance answers “how uncertain is that average experience?”

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 19

# Common Distributions and When They Fit

**Priority 7/10**

**Why this matters in C/C++**

Distributions are reusable shapes for uncertain behavior. Recognizing a Bernoulli, binomial, geometric, Poisson, uniform, normal, or exponential model can simplify simulation and analysis, but forcing the wrong distribution creates false confidence.

## Core ideas

- Bernoulli models one yes/no trial.
- Binomial counts successes in n independent trials with common success probability.
- Geometric counts trials until the first success under a memoryless Bernoulli model.
- Poisson often models counts of independent rare events over an interval.
- Exponential models waiting time in a memoryless event process.
- Normal distributions arise from many additive effects but are not universal.

**Core mathematical tools**

$$P(X = k) = \binom{n}{k} p^k (1-p)^{n-k}$$

$$E[\text{Binomial}] = np$$

$$P(\text{Poisson}(\lambda) = k) = e^{-\lambda}\lambda^k/k!$$

$$f_{\text{exp}}(t) = \lambda e^{-\lambda t}$$

**Where you will use it**

- load generation
- reliability tests
- queue simulations
- packet-arrival models
- Monte Carlo methods
- noise modeling

CHAPTER 19

# Common Distributions and When They Fit - Engineering Lab

**Priority 7/10**

## Worked example

If independent requests each time out with probability 0.002, the number of timeouts among 10,000 requests is binomial with mean 20. For small p and large n, a Poisson approximation with lambda=np=20 can be useful for rough planning.

## C++ connection

```cpp
#include <random>

int simulated_timeouts(std::mt19937_64& rng, int n, double p) {
   std::binomial_distribution<int> dist(n, p);
   return dist(rng);
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** load generation, reliability tests, queue simulations, packet-arrival models, Monte Carlo methods, noise modeling.

CHAPTER 19

# Common Distributions and When They Fit - Reliability Notes

**Priority 7/10**

**Failure modes**

- Assuming arrivals are Poisson because the formula is convenient.
- Using normal approximations for bounded or extremely skewed data.
- Treating samples from a PRNG as truly independent physical events.
- Forgetting parameter units such as events per second.
- Confusing a probability density with a point probability.

**Engineering checks**

- Justify why the distribution fits the mechanism.
- Compare empirical histogram and quantiles with the model.
- Keep distribution parameters in meaningful units.
- Use standard library distributions instead of ad-hoc transformations when suitable.

## Build intuition

Choose a plausible first model for each case: packet loss yes/no, number of failures in 1 hour, time until a memoryless failure, and additive sensor noise. State one reason each model could fail.

**Chapter takeaway**

A distribution is a model of a mechanism, not merely a curve that resembles a histogram.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 20

# Descriptive Statistics, Percentiles, and Correlation

**Priority 7/10**

**Why this matters in C/C++**

Engineers need compact summaries of measurements without destroying important structure. Latency, memory use, frame time, throughput, and benchmark results frequently require percentiles and robust statistics rather than one average.

## Core ideas

- The mean uses every value but is sensitive to outliers.
- The median is the 50th percentile and is robust to extreme values.
- A percentile answers what fraction of observations lie at or below a threshold.
- Interquartile range summarizes the middle half of data.
- Correlation measures linear association and does not prove causation.
- Rates and ratios often require weighted or harmonic means rather than a plain arithmetic mean.

**Core mathematical tools**

$$\bar{x} = \frac{1}{n}\sum_i x_i$$

$$\text{IQR} = Q_3 - Q_1$$

$$r = \frac{\text{Cov}(X,Y)}{\sigma_X \sigma_Y}$$

**Where you will use it**

- benchmark reports
- service latency
- frame-time analysis
- telemetry
- capacity planning
- regression detection

CHAPTER 20

# Descriptive Statistics, Percentiles, and Correlation - Engineering Lab

**Priority 7/10**

## Worked example

A service may have median latency 2 ms and 99th percentile 80 ms. Reporting only the mean can hide the user-visible tail. Percentiles describe how bad the slower fraction of requests can be.

## C++ connection

```cpp
#include <algorithm>
#include <vector>

double quantile(std::vector<double> x, double q) {
   if (x.empty()) return 0.0;
   q = std::clamp(q, 0.0, 1.0);
   std::sort(x.begin(), x.end());
   const double pos = q * (x.size() - 1);
   const auto i = static_cast<std::size_t>(pos);
   const auto j = std::min(i + 1, x.size() - 1);
   return x[i] + (pos - i) * (x[j] - x[i]);
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** benchmark reports, service latency, frame-time analysis, telemetry, capacity planning, regression detection.

CHAPTER 20

# Descriptive Statistics, Percentiles, and Correlation - Reliability Notes

**Priority 7/10**

**Failure modes**

- Comparing percentile values computed with different conventions.
- Averaging percentiles from separate machines.
- Using correlation as evidence of causation.
- Ignoring multimodal data.
- Reporting too many digits from noisy measurements.

**Engineering checks**

- State the percentile convention or use one tool consistently.
- Aggregate raw samples or merge distributions carefully.
- Include sample count and measurement conditions.
- Visualize histograms or empirical CDFs when tails matter.

## Build intuition

Given latencies [1,1,1,2,2,3,4,20,40,100] ms, compute the mean and median. Explain which better describes a typical request and which exposes total cost.

**Chapter takeaway**

Good statistics preserve the shape of engineering reality instead of compressing it into a misleading single number.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 21

# Sampling, Confidence Intervals, and Benchmark Uncertainty

**Priority 8/10**

**Why this matters in C/C++**

A benchmark is a sample from a noisy process. Compiler warm-up, CPU frequency, caches, interrupts, OS scheduling, and input variation all affect measurements. Statistics helps distinguish a real speedup from noise.

## Core ideas

- A sample estimates properties of a larger process or population.
- Standard error describes uncertainty in an estimated mean under a model.
- A confidence interval is a procedure with repeated-sampling coverage, not a probability that a fixed parameter moves around.
- Larger independent samples usually narrow uncertainty roughly with 1/sqrt(n).
- Paired measurements can reduce noise when two variants are tested under matched conditions.
- Effect size matters in addition to statistical significance.

**Core mathematical tools**

$$SE(\bar{x}) = \frac{s}{\sqrt{n}}$$

rough 95% interval $\approx \bar{x} \pm 1.96\,SE$ under suitable conditions

$$\text{speedup} = \frac{T_{old}}{T_{new}}$$

**Where you will use it**

- microbenchmarks
- A/B performance experiments
- hardware evaluation
- compiler optimization validation
- latency regression testing

CHAPTER 21

# Sampling, Confidence Intervals, and Benchmark Uncertainty - Engineering Lab

**Priority 8/10**

## Worked example

If two benchmark versions differ by 1 percent but run-to-run variation is 5 percent, one run proves almost nothing. Repeated randomized or paired runs, warm-up, and a confidence interval can show whether the difference is stable enough to act on.

## C++ connection

```cpp
#include <chrono>

template<class F>
auto elapsed(F&& f) {
   const auto t0 = std::chrono::steady_clock::now();
   f();
   const auto t1 = std::chrono::steady_clock::now();
   return t1 - t0;
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** microbenchmarks, A/B performance experiments, hardware evaluation, compiler optimization validation, latency regression testing.

CHAPTER 21

# Sampling, Confidence Intervals, and Benchmark Uncertainty - Reliability Notes

**Priority 8/10**

**Failure modes**

- Taking only the fastest run and calling it typical.
- Benchmarking code the optimizer removes.
- Using tiny timings near clock resolution.
- Comparing means without looking at variance or tails.
- Treating repeated measurements from one warmed cache state as independent.

**Engineering checks**

- Use a benchmark harness and consume results so work is observable.
- Randomize variant order when practical.
- Report sample count, central tendency, spread, and environment.
- Separate measurement error from real workload variation.

## Build intuition

Version A takes 100.0 +/- 3.0 units and version B takes 98.8 +/- 3.2 across repeated runs. Explain why “B is 1.2 percent faster” is not yet a strong engineering conclusion.

**Chapter takeaway**

Performance numbers are measurements with uncertainty, not exact constants.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 22

# Hash Collisions, Bloom Filters, and Probabilistic Data Structures

**Priority 8/10**

**Why this matters in C/C++**

Hashing turns large keys into compact values, so collisions are mathematically unavoidable. Probabilistic structures deliberately trade small error probabilities for large memory or speed improvements.

## Core ideas

- A hash maps a large key space into a finite bucket space.
- The birthday effect makes collision probability grow faster than naive intuition.
- Load factor influences expected collision behavior in hash tables.
- Bloom filters can have false positives but no false negatives when used correctly.
- Multiple hash positions reduce Bloom-filter false positives up to an optimal point.
- Adversarial input changes the assumptions behind expected hash-table behavior.

**Core mathematical tools**

$$\alpha = \frac{n}{m} \text{ (load factor)}$$

$$p_{\text{Bloom}} \approx (1 - e^{-kn/m})^k$$

$$k_{opt} \approx \frac{m}{n} \ln 2$$

**Where you will use it**

- unordered containers
- deduplication
- caches
- storage engines
- network filters
- database membership tests

CHAPTER 22

# Hash Collisions, Bloom Filters, and Probabilistic Data Structures - Engineering Lab

**Priority 8/10**

## Worked example

For a Bloom filter with m bits, n inserted keys, and k hash probes, the false-positive rate can be estimated before allocating memory. This allows a direct engineering trade: bytes of memory versus extra downstream lookups.

## C++ connection

```cpp
#include <cstddef>
#include <cmath>

double bloom_fp(std::size_t m, std::size_t n, std::size_t k) {
   if (m == 0) return 1.0;
   const double x = std::exp(-double(k) * double(n) / double(m));
   return std::pow(1.0 - x, double(k));
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** unordered containers, deduplication, caches, storage engines, network filters, database membership tests.

CHAPTER 22

# Hash Collisions, Bloom Filters, and Probabilistic Data Structures - Reliability Notes

**Priority 8/10**

**Failure modes**

- Assuming hash outputs are independent and uniform without evidence.
- Using a non-cryptographic hash against attacker-controlled keys.
- Forgetting Bloom-filter saturation as n grows beyond design capacity.
- Confusing false-positive probability with probability that a particular returned key is wrong.
- Using modulo bias in bucket selection.

**Engineering checks**

- Track actual load factor.
- Reserve expected unordered-container size when known.
- Choose hash strategy based on threat model.
- Test Bloom-filter false-positive rate at and beyond planned capacity.

## Build intuition

Design a Bloom filter for one million keys with 10 bits per key. Use the approximate optimal k and estimate the false-positive probability.

**Chapter takeaway**

Probabilistic structures are controlled approximations: make the error budget explicit and connect it to memory and latency budgets.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

Part IV

# Machine Arithmetic and Numerical Reliability

This part is critical for C and C++ because the languages expose integer widths, conversion rules, floating-point behavior, and low-level representation directly. Mathematical correctness and machine correctness are not automatically the same thing.

CHAPTER 23

# Fixed-Width Integers, Ranges, Promotions, and Overflow

**Priority 9/10**

**Why this matters in C/C++**

Integer arithmetic looks exact, but the representable range is finite and C/C++ conversion rules can change both value and signedness before an expression is evaluated. This is a major source of security and systems bugs.

## Core ideas

- An N-bit unsigned integer represents 0 through 2^N-1 and arithmetic wraps modulo 2^N.
- Signed integer overflow in C and C++ is undefined behavior; do not use it as intentional wrap-around.
- Integer promotions may widen small integer types before arithmetic.
- Mixed signed/unsigned comparisons can convert the signed operand to an unsigned type.
- Multiplication can overflow before a later division reduces the mathematical result.
- Range reasoning belongs in API contracts and tests, not only in comments.

**Core mathematical tools**

$$0 \le u \le 2^N - 1$$

$$u + v \equiv u + v \pmod{2^N} \text{ for N-bit unsigned}$$

$|a \times b| \le$ max must hold before storing a signed product

**Where you will use it**

- parsers and binary formats
- allocators and size calculations
- cryptographic primitives
- embedded register code
- serialization
- security-critical bounds checks

CHAPTER 23

# Fixed-Width Integers, Ranges, Promotions, and Overflow - Engineering Lab

**Priority 9/10**

## Worked example

Suppose count and element_size are individually valid size_t values. The mathematical product can still exceed SIZE_MAX. A bounds check that happens after multiplication is too late because the unsigned product has already wrapped.

## C++ connection

```cpp
#include <cstddef>
#include <limits>
#include <optional>

std::optional<std::size_t> checked_mul(std::size_t a, std::size_t b) {
   if (a != 0 && b > std::numeric_limits<std::size_t>::max() / a)
      return std::nullopt;
   return a * b;
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** parsers and binary formats, allocators and size calculations, cryptographic primitives, embedded register code, serialization, security-critical bounds checks.

CHAPTER 23

# Fixed-Width Integers, Ranges, Promotions, and Overflow - Reliability Notes

**Priority 9/10**

**Failure modes**

- Checking a*b > limit after a*b has already overflowed.
- Mixing int with size_t in comparisons.
- Narrowing without validating the destination range.
- Relying on signed overflow to wrap.
- Shifting by a negative count or by at least the promoted width.

**Engineering checks**

- Use fixed-width or size-related types intentionally.
- Check before arithmetic that can overflow.
- Compile with warnings and sanitizers during development.
- Test maximum, minimum, zero, and one-past-boundary cases.

## Build intuition

For 32-bit unsigned values, determine whether 100000*50000 fits. Then design a precondition that detects overflow without computing the overflowing product.

**Chapter takeaway**

Machine integer arithmetic has a finite algebra. Write code for that algebra, not for unbounded mathematical integers.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 24

# Fixed-Point Arithmetic and Q Formats

**Priority 9/10**

**Why this matters in C/C++**

Embedded, DSP, finance-like deterministic calculations, and hardware interfaces sometimes need fixed-point arithmetic because floating point is unavailable, too costly, or semantically undesirable.

## Core ideas

- A fixed-point number stores an integer scaled by a constant factor.
- In binary Q formats the scaling factor is a power of two.
- Addition requires matching scales; multiplication doubles the fractional-bit count before rescaling.
- Rounding policy is part of the numerical contract.
- Saturation and wrap-around represent different arithmetic models.
- Range and resolution trade against each other for a fixed bit width.

**Core mathematical tools**

$$x_{real} = \frac{x_{raw}}{2^F}$$

$$\Delta = 2^{-F}$$

$$(a \times b)_{raw} \approx \frac{a_{raw} b_{raw}}{2^F}$$

**Where you will use it**

- microcontrollers
- audio DSP
- motor control
- deterministic simulation
- packed numeric formats

CHAPTER 24

# Fixed-Point Arithmetic and Q Formats - Engineering Lab

**Priority 9/10**

## Worked example

In Q16.16, the raw integer 98304 represents 98304/65536 = 1.5. Multiplying two Q16.16 values requires a wider intermediate, then a 16-bit right shift with a chosen rounding rule.

## C++ connection

```cpp
#include <cstdint>

std::int32_t q16_mul(std::int32_t a, std::int32_t b) {
   const std::int64_t wide = std::int64_t{a} * std::int64_t{b};
   return static_cast<std::int32_t>(wide >> 16); // truncating example
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** microcontrollers, audio DSP, motor control, deterministic simulation, packed numeric formats.

CHAPTER 24

# Fixed-Point Arithmetic and Q Formats - Reliability Notes

**Priority 9/10**

**Failure modes**

- Multiplying in the narrow storage type.
- Forgetting that right shift of negative signed values is representation/implementation sensitive across some contexts and standards histories.
- Mixing different Q formats silently.
- Ignoring saturation requirements.
- Assuming truncation and round-to-nearest are interchangeable.

**Engineering checks**

- Document total bits, sign convention, and fractional bits in the type.
- Use a wider intermediate.
- Test values around 0, 1, -1, maximum, and minimum.
- Quantify worst-case quantization error.

## Build intuition

Design a signed 16-bit fixed-point format that must cover at least [-8, 8) with the best power-of-two resolution. How many fractional bits can it use?

**Chapter takeaway**

Fixed point is integer arithmetic plus an explicit scale; once the scale is visible, the design becomes predictable.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 25

# IEEE 754 Floating Point: Representation, Rounding, and Special Values

**Priority 10/10**

**Why this matters in C/C++**

Floating point is essential in scientific, graphics, simulation, signal, ML, and geometry code. It is also a common source of wrong intuition. The central fact is simple: most real numbers cannot be represented exactly in a finite binary format.

## Core ideas

- A binary floating value is built from sign, significand, and exponent.
- Normalized values provide roughly constant relative precision across a wide range.
- Subnormal values extend gradual underflow near zero with reduced precision.
- Infinity and NaN encode exceptional numerical states.
- Rounding chooses a representable neighbor after an exact mathematical operation.
- Machine epsilon is related to spacing near 1, not a universal tolerance for every value.

**Core mathematical tools**

$$x = (-1)^s \times m \times 2^e$$

$\mathrm{fl}(x \circ y) = (x \circ y)(1+\delta),\ |\delta| \lesssim u$ for many ordinary cases

relative spacing grows with magnitude

**Where you will use it**

- scientific computing
- 3D graphics
- physics
- statistics
- machine learning
- DSP

CHAPTER 25

# IEEE 754 Floating Point: Representation, Rounding, and Special Values - Engineering Lab

**Priority 10/10**

## Worked example

The decimal value 0.1 has no finite binary fractional representation, just as 1/3 has no finite decimal representation. A double stores the nearest representable binary value. Therefore 0.1+0.2 need not equal the nearest representation of decimal 0.3 bit-for-bit after separate roundings.

## C++ connection

```cpp
#include <cmath>
#include <limits>
#include <iostream>

void inspect_fp() {
   std::cout << std::numeric_limits<double>::digits << " bits
";
   std::cout << std::numeric_limits<double>::epsilon() << '\n';
   std::cout << std::isfinite(1.0 / 0.0) << '\n';
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** scientific computing, 3D graphics, physics, statistics, machine learning, DSP.

CHAPTER 25

# IEEE 754 Floating Point: Representation, Rounding, and Special Values - Reliability Notes

**Priority 10/10**

**Failure modes**

- Using epsilon as a fixed tolerance for values of every magnitude.
- Assuming NaN compares equal to itself.
- Ignoring signed zero in branch-sensitive numerical code.
- Believing decimal text round-trips with arbitrary formatting precision.
- Changing compiler floating-point modes without understanding reassociation and exception implications.

**Engineering checks**

- Use numeric_limits to inspect implementation properties.
- Test finite, infinity, NaN, zero, subnormal-near cases where relevant.
- Choose absolute and relative tolerances from problem scale.
- Specify required floating semantics for reproducible or safety-critical software.

## Build intuition

Explain why the gap between adjacent doubles near 1e20 is much larger in absolute terms than the gap near 1.0, even though both have similar relative precision.

**Chapter takeaway**

Floating point is not “broken real arithmetic.” It is a carefully defined finite approximation system with its own geometry of representable numbers.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 26

# Floating-Point Comparison, Cancellation, and Stable Summation

**Priority 10/10**

**Why this matters in C/C++**

Correct formulas can still produce poor numerical answers. C/C++ engineers need practical patterns for comparison, cancellation avoidance, summation, and stable algebraic rearrangement.

## Core ideas

- Absolute tolerance is useful near zero; relative tolerance scales with magnitude.
- Catastrophic cancellation occurs when nearly equal values are subtracted and leading digits disappear.
- Order of summation affects floating results.
- Compensated summation tracks low-order information lost by ordinary addition.
- Specialized functions such as log1p, expm1, hypot, and fma often improve accuracy and range behavior.
- A stable algorithm controls error growth rather than merely using a wider type.

**Core mathematical tools**

$$|a-b| \le \varepsilon_{abs} + \varepsilon_{rel}\,\max(|a|,|b|)$$

cancellation: $x-y$ is fragile when $x \approx y$

**Where you will use it**

- geometry predicates
- scientific reductions
- probability calculations
- financial approximations
- simulation

CHAPTER 26

# Floating-Point Comparison, Cancellation, and Stable Summation - Engineering Lab

**Priority 10/10**

## Worked example

Summing one million tiny values after a huge value can lose contributions because each small addend is below the local spacing. Kahan-style compensation stores part of the rounding loss and can dramatically improve a long reduction.

## C++ connection

```cpp
#include <span>

double kahan_sum(std::span<const double> x) {
   double sum = 0.0, c = 0.0;
   for (double v : x) {
      double y = v - c;
      double t = sum + y;
      c = (t - sum) - y;
      sum = t;
   }
   return sum;
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** geometry predicates, scientific reductions, probability calculations, financial approximations, simulation.

CHAPTER 26

# Floating-Point Comparison, Cancellation, and Stable Summation - Reliability Notes

**Priority 10/10**

**Failure modes**

- Comparing doubles with == when values come from independent computations.
- Using a tolerance without considering units.
- Subtracting nearly equal squared lengths.
- Summing a long vector in arbitrary order when reproducibility matters.
- Assuming long double is portable in precision and format.

**Engineering checks**

- Derive tolerance from domain scale.
- Use stable library functions for near-zero transformations.
- Compare naive and compensated reductions on adversarial data.
- Document reproducibility requirements across threads and architectures.

## Build intuition

Construct values a and b near 1e16 whose difference is small. Explain why forming a+b and later subtracting a can lose information about b.

**Chapter takeaway**

Numerical reliability is an algorithm property, not just a data-type choice.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 27

# Conditioning, Error Propagation, and Numerical Stability

**Priority 10/10**

**Why this matters in C/C++**

Some problems amplify tiny input errors no matter how carefully they are programmed; other problems are well-conditioned but can be ruined by a poor algorithm. Distinguishing conditioning from stability is central to serious numerical C/C++.

## Core ideas

- Conditioning describes sensitivity of the mathematical problem to input perturbations.
- Stability describes how much additional error an algorithm introduces.
- Forward error compares computed output with the exact output.
- Backward error asks what nearby input would make the computed result exact.
- Relative error is often more meaningful than absolute error across scales.
- Error propagation can be estimated with derivatives for small perturbations.

**Core mathematical tools**

$$\text{absolute error} = |\hat{x} - x|$$

$$\text{relative error} = \frac{|\hat{x} - x|}{|x|}$$

$$\delta f \approx f'(x)\,\delta x$$

**Where you will use it**

- linear solvers
- simulation
- geometry
- calibration
- scientific libraries
- sensor processing

CHAPTER 27

# Conditioning, Error Propagation, and Numerical Stability - Engineering Lab

**Priority 10/10**

## Worked example

The function f(x)=1/(1-x) becomes highly sensitive as x approaches 1. Even a tiny measurement or rounding error in x can cause a large output change. No clever implementation can remove the underlying conditioning; the software must recognize and manage the sensitive region.

## C++ connection

```cpp
#include <cmath>
#include <optional>

std::optional<double> reciprocal_gap(double x) {
   const double d = 1.0 - x;
   if (std::abs(d) < 1e-12) return std::nullopt;
   return 1.0 / d;
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** linear solvers, simulation, geometry, calibration, scientific libraries, sensor processing.

CHAPTER 27

# Conditioning, Error Propagation, and Numerical Stability - Reliability Notes

**Priority 10/10**

**Failure modes**

- Blaming floating point for an inherently ill-conditioned problem.
- Reporting many digits unsupported by input accuracy.
- Using relative error at an exact zero without a separate rule.
- Ignoring scale differences between variables.
- Validating only against outputs generated by the same algorithm.

**Engineering checks**

- Estimate sensitivity before promising output accuracy.
- Use higher-precision reference results in tests.
- Perturb inputs slightly and observe output sensitivity.
- Separate model uncertainty, input error, and rounding error.

## Build intuition

For f(x)=x^2, use a derivative to estimate the output error near x=100 when the input uncertainty is +/-0.01. Compare absolute and relative errors.

**Chapter takeaway**

You cannot code away bad conditioning, but you can detect it, choose stable algorithms, and report meaningful accuracy.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 28

# Root Finding, Interpolation, and Numerical Integration

**Priority 9/10**

**Why this matters in C/C++**

Many engineering problems reduce to finding where a function crosses zero, estimating values between samples, or accumulating an area. These operations appear in calibration, simulation, geometry, physics, and signal processing.

## Core ideas

- Bisection is slow but robust when a continuous function changes sign across an interval.
- Newton iteration can converge rapidly but needs a derivative and a good starting point.
- Secant methods approximate derivative information from values.
- Linear interpolation is local and simple; higher-order interpolation can oscillate.
- The trapezoidal rule approximates an integral from sampled values.
- Adaptive algorithms spend work where the function is difficult.

**Core mathematical tools**

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

$$\int_a^b f(x)dx \approx \frac{h}{2}[f(a) + 2\sum f(x_i) + f(b)]$$

**Where you will use it**

- physics simulation
- calibration curves
- ray intersections
- control tuning
- scientific computing
- signal energy estimation

CHAPTER 28

# Root Finding, Interpolation, and Numerical Integration - Engineering Lab

**Priority 9/10**

## Worked example

To find sqrt(2), solve f(x)=x^2-2=0. Bisection on [1,2] halves the uncertainty every iteration. Newton iteration x <- 0.5(x+2/x) converges much faster near the root but needs division and a sensible non-zero start.

## C++ connection

```cpp
#include <cmath>

double sqrt_newton(double a) {
   if (a <= 0.0) return 0.0;
   double x = a >= 1.0 ? a : 1.0;
   for (int i = 0; i < 20; ++i)
      x = 0.5 * (x + a / x);
   return x;
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** physics simulation, calibration curves, ray intersections, control tuning, scientific computing, signal energy estimation.

CHAPTER 28

# Root Finding, Interpolation, and Numerical Integration - Reliability Notes

**Priority 9/10**

**Failure modes**

- Running Newton iteration without a convergence or iteration limit.
- Ignoring multiple roots or zero derivatives.
- Using high-order interpolation across discontinuities.
- Assuming smaller integration steps always justify the extra cost.
- Stopping based only on absolute change regardless of scale.

**Engineering checks**

- Use bracketing when robustness matters.
- Set iteration and domain guards.
- Compare against analytic cases.
- Use absolute+relative convergence criteria.

## Build intuition

How many bisection iterations are needed to shrink an initial interval of width 1 to below 1e-9? Express the answer using a base-2 logarithm.

**Chapter takeaway**

Numerical methods trade iterations for accuracy. Robustness comes from knowing the assumptions of the method and guarding them in code.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

Part V

# Systems, Memory, Performance, and Concurrency

C and C++ are dominant where hardware behavior matters. This part connects arithmetic to addresses, caches, throughput, parallelism, and deadlines so that performance discussions become quantitative rather than intuitive.

CHAPTER 29

# Address Arithmetic, Alignment, Padding, and Layout

**Priority 9/10**

**Why this matters in C/C++**

Memory layout is mathematics over byte addresses. Alignment, padding, strides, object representation, and array indexing all reduce to integer offsets and divisibility constraints.

## Core ideas

- An aligned address is divisible by the required alignment.
- Power-of-two alignment can be rounded up with a mask when overflow is controlled.
- Array element i is base + i*stride in bytes.
- Structure size can include padding between members and at the end.
- Multidimensional row-major indexing is a mixed-radix calculation.
- Pointer arithmetic in C/C++ is constrained by object and array rules; raw numeric address arithmetic is not a portable substitute.

**Core mathematical tools**

$$\text{aligned}(x, A) \iff x \bmod A = 0$$

$$\text{roundUp}(x, A) = \left\lceil \frac{x}{A} \right\rceil A$$

$$\text{offset}_{2D} = (r \cdot cols + c) \cdot \text{sizeof}(T)$$

**Where you will use it**

- allocators
- SIMD buffers
- binary formats
- memory-mapped devices
- image strides
- cache-aware structures

CHAPTER 29

# Address Arithmetic, Alignment, Padding, and Layout - Engineering Lab

**Priority 9/10**

## Worked example

For 64-byte alignment, an address offset 130 must be rounded to 192. With power-of-two A=64, (130+63)&~63 performs the integer ceiling-to-multiple operation, provided the addition cannot overflow.

## C++ connection

```cpp
#include <cstddef>
#include <optional>
#include <limits>

std::optional<std::size_t> align_up(std::size_t x, std::size_t a) {
   if (a == 0 || (a & (a - 1)) != 0) return std::nullopt;
   if (x > std::numeric_limits<std::size_t>::max() - (a - 1))
      return std::nullopt;
   return (x + a - 1) & ~(a - 1);
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** allocators, SIMD buffers, binary formats, memory-mapped devices, image strides, cache-aware structures.

CHAPTER 29

# Address Arithmetic, Alignment, Padding, and Layout - Reliability Notes

**Priority 9/10**

**Failure modes**

- Assuming sizeof(struct) equals the sum of member sizes.
- Overflowing base+count*stride.
- Using bit-mask alignment for a non-power-of-two value.
- Ignoring array padding/stride from external APIs.
- Converting arbitrary integers to pointers and assuming portable semantics.

**Engineering checks**

- Use offsetof/sizeof to inspect actual layouts when legal and relevant.
- Check multiplication and addition separately.
- Assert alignment requirements at API boundaries.
- Test odd widths and padded strides for image or tensor code.

## Build intuition

A 1920x1080 RGB image uses a 2048-pixel row stride with 4 bytes per pixel. Compute the allocated byte size and explain why width*height*4 is insufficient.

**Chapter takeaway**

Memory layout becomes easier when every address is treated as base plus a checked, unit-correct offset.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 30

# Caches, Locality, Bandwidth, and Arithmetic Intensity

**Priority 9/10**

**Why this matters in C/C++**

Modern C++ performance is often limited by data movement rather than arithmetic. Cache lines, working sets, spatial locality, temporal locality, and memory bandwidth are numerical constraints on algorithm design.

## Core ideas

- Spatial locality rewards nearby accesses; temporal locality rewards reuse.
- A cache line transfers a block, not one scalar.
- Working-set size relative to cache capacity strongly affects hit rate.
- Bandwidth measures bytes per second; latency measures time for a dependency to complete.
- Arithmetic intensity is useful work per byte moved.
- Data layout can change effective bandwidth without changing Big-O complexity.

**Core mathematical tools**

$$\text{bandwidth} = \frac{\text{bytes moved}}{\text{time}}$$

$$\text{arithmetic intensity} = \frac{\text{operations}}{\text{bytes transferred}}$$

$$\text{time}_{min} \gtrsim \frac{\text{bytes}}{\text{sustained bandwidth}}$$

**Where you will use it**

- HPC kernels
- game engines
- databases
- image processing
- allocators
- data-oriented design

CHAPTER 30

# Caches, Locality, Bandwidth, and Arithmetic Intensity - Engineering Lab

**Priority 9/10**

## Worked example

Scanning 1 GiB once on a system sustaining 50 GiB/s cannot take less than about 20 ms purely from memory traffic, before other effects. This lower bound immediately tells you whether a claimed 2 ms full scan is plausible.

## C++ connection

```cpp
#include <span>

double dot(std::span<const float> a, std::span<const float> b) {
   double s = 0.0;
   for (std::size_t i = 0; i < a.size(); ++i)
      s += double(a[i]) * double(b[i]);
   return s; // contiguous streams help hardware prefetching
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** HPC kernels, game engines, databases, image processing, allocators, data-oriented design.

CHAPTER 30

# Caches, Locality, Bandwidth, and Arithmetic Intensity - Reliability Notes

**Priority 9/10**

**Failure modes**

- Counting only explicit loads and forgetting write-allocate or cache effects.
- Assuming peak vendor bandwidth equals sustained application bandwidth.
- Using linked structures for scan-heavy workloads without measuring locality.
- Confusing latency with bandwidth.
- Optimizing arithmetic while the kernel is memory-bound.

**Engineering checks**

- Calculate bytes moved per logical element.
- Measure working-set size at several problem sizes.
- Use hardware counters when available.
- Compare array-of-structures and structure-of-arrays based on actual access patterns.

## Build intuition

A kernel reads two float arrays and writes one float array while performing two arithmetic operations per element. Estimate its arithmetic intensity ignoring cache reuse.

**Chapter takeaway**

Performance starts with a traffic budget: how many bytes must move, how often, and from where?

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 31

# Latency, Throughput, Amdahl, Gustafson, and Little's Law

**Priority 10/10**

**Why this matters in C/C++**

System performance uses several distinct quantities. Latency is time per operation, throughput is completed work per time, and parallel speedup depends on both serial fractions and available concurrency.

## Core ideas

- Latency and throughput are related but not identical; pipelining can raise throughput without reducing single-operation latency.
- Amdahl's law limits fixed-workload speedup by the serial fraction.
- Gustafson's law explains why larger workloads can use more processors efficiently.
- Little's law connects average items in a stable system with throughput and average time in system.
- Utilization near 100 percent can cause large queueing delays.
- Critical-path latency is governed by dependent work, not total independent work.

**Core mathematical tools**

$$S_{Amdahl} = \frac{1}{(1-p) + p/N}$$

$$S_{Gustafson} = N - (1-p)(N-1)$$

$$L = \lambda W$$

**Where you will use it**

- servers
- parallel algorithms
- pipelines
- task systems
- GPU/CPU throughput reasoning
- queue sizing

CHAPTER 31

# Latency, Throughput, Amdahl, Gustafson, and Little’s Law - Engineering Lab

**Priority 10/10**

## Worked example

If 5 percent of a workload is inherently serial, even infinitely many processors cap Amdahl speedup at 20x. With 16 processors, the bound is about 9.14x before synchronization or memory effects.

## C++ connection

```cpp
#include <cmath>

double amdahl(double parallel_fraction, unsigned workers) {
   if (workers == 0) return 0.0;
   return 1.0 / ((1.0 - parallel_fraction) +
               parallel_fraction / workers);
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** servers, parallel algorithms, pipelines, task systems, GPU/CPU throughput reasoning, queue sizing.

CHAPTER 31

# Latency, Throughput, Amdahl, Gustafson, and Little's Law - Reliability Notes

**Priority 10/10**

**Failure modes**

- Using 1/latency as throughput when operations overlap.
- Claiming linear speedup without including serial and synchronization work.
- Applying Little's law to an unstable or ill-defined interval.
- Ignoring queueing delay at high utilization.
- Confusing CPU utilization with useful work.

**Engineering checks**

- Measure latency distributions and throughput separately.
- Estimate serial fraction before adding threads.
- Track queue depth and arrival/service rates.
- Distinguish service time from total response time.

## Build intuition

A service completes 20,000 requests/s and contains an average of 300 in-flight requests. Under stable conditions, use Little's law to estimate average time in the system.

**Chapter takeaway**

Performance engineering improves when every claim is attached to the correct quantity: latency, throughput, concurrency, or utilization.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 32

# SIMD, Vector Width, Reductions, and Data-Parallel Math

**Priority 10/10**

**Why this matters in C/C++**

SIMD lets one instruction operate on several lanes, but vector width alone does not guarantee proportional speedup. C/C++ engineers need the arithmetic of lanes, tails, alignment, reductions, and memory traffic.

## Core ideas

- A W-bit vector holds W/b lanes of b-bit elements.
- Vectorization benefits independent, regular work.
- Tail elements need masking, a scalar cleanup, or padded data.
- Reductions introduce dependencies across lanes and may change floating-point order.
- Wider vectors can increase throughput while also changing frequency, power, or instruction availability on some processors.
- The limiting resource can be memory bandwidth rather than vector ALUs.

**Core mathematical tools**

$$\text{lanes} = \frac{\text{vector bits}}{\text{element bits}}$$

$$\text{ideal vector work per instruction} \approx \text{lanes}$$

$$\text{speedup} \leq \min(\text{compute gain}, \text{bandwidth headroom}, \ldots)$$

**Where you will use it**

- image/video
- DSP
- HPC
- physics
- cryptography
- ML inference

CHAPTER 32

# SIMD, Vector Width, Reductions, and Data-Parallel Math - Engineering Lab

**Priority 10/10**

## Worked example

A 256-bit vector holds eight 32-bit floats. Processing 1003 floats therefore has 125 full vectors plus a 3-element tail. The tail is part of the algorithm, not an afterthought.

## C++ connection

```cpp
#include <cstddef>

float scalar_sum(const float* x, std::size_t n) {
   float s = 0.0f;
   for (std::size_t i = 0; i < n; ++i) s += x[i];
   return s; // compilers may auto-vectorize; reassociation affects FP
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** image/video, DSP, HPC, physics, cryptography, ML inference.

CHAPTER 32

# SIMD, Vector Width, Reductions, and Data-Parallel Math - Reliability Notes

**Priority 10/10**

**Failure modes**

- Assuming AVX-512 means exactly 2x AVX2 application speed.
- Ignoring tail handling.
- Using aligned loads on unaligned data.
- Expecting vectorization across loop-carried dependencies.
- Demanding bit-identical floating reductions after reassociation.

**Engineering checks**

- Check compiler vectorization reports or generated code.
- Benchmark representative sizes including non-multiples of lane count.
- Measure bandwidth as well as instructions.
- State whether reproducible reduction order is required.

## Build intuition

For 512-bit vectors operating on doubles, compute lane count. For an array of 10,000 elements, calculate the number of complete vectors and remaining tail elements.

**Chapter takeaway**

SIMD is parallel arithmetic with structural constraints. Count lanes, dependencies, bytes, and tails before predicting speedup.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 33

# Concurrency Math: Partial Orders, Contention, False Sharing, and Real-Time Timing

**Priority 10/10**

**Why this matters in C/C++**

Threads introduce ordering rather than just simultaneous execution. Atomics, happens-before relations, contention, cache-line sharing, and deadlines all require mathematical models of time and order.

## Core ideas

- A partial order allows some events to be ordered while others remain incomparable.
- Synchronization creates ordering constraints between operations.
- Contention increases waiting as more actors compete for one serialized resource.
- False sharing occurs when independent variables share a coherency unit such as a cache line.
- Periodic real-time tasks are described by execution time, period, and deadline.
- The least common multiple of task periods gives a repeating hyperperiod when periods are integral.

**Core mathematical tools**

$$U = \sum_i \frac{C_i}{T_i}$$

$$H = \mathrm{lcm}(T_1, T_2, \ldots)$$

speedup is limited by serialized critical sections

**Where you will use it**

- thread pools
- lock-free structures
- real-time embedded C++
- parallel runtimes
- high-frequency services

CHAPTER 33

# Concurrency Math: Partial Orders, Contention, False Sharing, and Real-Time Timing - Engineering Lab

**Priority 10/10**

## Worked example

Two counters updated by different threads may still fight over one cache line. The variables are logically independent, but hardware coherence serializes ownership of the line. Padding or per-thread aggregation can transform the performance mathematics.

## C++ connection

```cpp
#include <atomic>
#include <cstddef>

struct alignas(64) Counter {
   std::atomic<std::uint64_t> value{0};
};

// Separate frequently-written counters to reduce false sharing.
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** thread pools, lock-free structures, real-time embedded C++, parallel runtimes, high-frequency services.

CHAPTER 33

# Concurrency Math: Partial Orders, Contention, False Sharing, and Real-Time Timing - Reliability Notes

**Priority 10/10**

**Failure modes**

- Equating source-code order with inter-thread visibility.
- Using relaxed atomics without a proven ordering argument.
- Padding based on an assumed cache-line size without considering portability.
- Ignoring lock convoying and queueing.
- Treating utilization <100 percent as a complete schedulability proof.

**Engineering checks**

- Draw the required happens-before relationships.
- Stress-test with many interleavings and sanitizers.
- Measure contention as worker count increases.
- For real-time work, analyze worst-case execution time and deadlines, not average time.

## Build intuition

Three periodic tasks have (C,T) pairs (1,5), (2,10), and (4,20) ms. Compute total utilization and the hyperperiod. Explain what the numbers do and do not prove.

**Chapter takeaway**

Concurrency is mathematics of order, shared capacity, and time. Correctness comes before parallel speedup.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

Part VI

# Linear Algebra, Geometry, and Spatial Computing

Vectors and matrices are the common language of graphics, games, robotics, simulation, computer vision, numerical solvers, and machine learning. The presentation stays concrete: every operation is attached to a geometric or computational meaning.

CHAPTER 34

# Vectors: Length, Dot Product, Cross Product, and Projection

**Priority 9/10**

**Why this matters in C/C++**

Vectors represent direction, displacement, velocity, force, features, colors, and many other multi-component quantities. Their basic operations are among the highest-value mathematical tools in C++ graphics, game, robotics, and HPC code.

## Core ideas

- A vector has components relative to a coordinate basis.
- Its Euclidean norm measures length.
- The dot product measures alignment and produces a scalar.
- A normalized vector has length 1 and is useful for direction-only calculations.
- Projection extracts the component of one vector along another.
- In 3D, the cross product produces a vector perpendicular to two inputs.

**Core mathematical tools**

$$\|v\| = \sqrt{v \cdot v}$$

$$a \cdot b = \sum_i a_i b_i = \|a\|\|b\| \cos\theta$$

$$\text{proj}_b(a) = \frac{a \cdot b}{b \cdot b} b$$

$$a \times b \perp a, b$$

**Where you will use it**

- lighting
- collision response
- robot motion
- camera geometry
- HPC kernels
- ML feature operations

CHAPTER 34

# Vectors: Length, Dot Product, Cross Product, and Projection - Engineering Lab

**Priority 9/10**

## Worked example

If a unit surface normal n and a velocity v have v dot n < 0, the velocity points into the surface. The normal component is (v dot n)n, so a simple frictionless reflection is v - 2(v dot n)n.

## C++ connection

```cpp
#include <cmath>

struct Vec3 { double x, y, z; };

double dot(Vec3 a, Vec3 b) { return a.x*b.x + a.y*b.y + a.z*b.z; }
Vec3 scale(Vec3 v, double s) { return {v.x*s,v.y*s,v.z*s}; }
Vec3 sub(Vec3 a, Vec3 b) { return {a.x-b.x,a.y-b.y,a.z-b.z}; }
Vec3 reflect(Vec3 v, Vec3 unit_n) {
   return sub(v, scale(unit_n, 2.0 * dot(v, unit_n)));
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** lighting, collision response, robot motion, camera geometry, HPC kernels, ML feature operations.

CHAPTER 34

# Vectors: Length, Dot Product, Cross Product, and Projection - Reliability Notes

**Priority 9/10**

**Failure modes**

- Normalizing a near-zero vector.
- Assuming the dot product itself is an angle.
- Using a non-unit normal in a formula that assumes unit length.
- Mixing points and direction vectors semantically.
- Losing precision in large-coordinate subtract-then-dot calculations.

**Engineering checks**

- Check norm before normalization.
- Encode point/vector distinctions in types when useful.
- Test orthogonal, parallel, anti-parallel, and zero cases.
- Use std::hypot-style scaling ideas when extreme magnitudes are possible.

## Build intuition

Let a=(3,4,0) and b=(1,0,0). Compute |a|, a dot b, the projection of a onto b, and the cosine of the angle between them.

**Chapter takeaway**

Dot, norm, projection, and cross product are small operations that unlock a large fraction of spatial programming.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 35

# Matrices, Linear Systems, Rank, and Decompositions

**Priority 9/10**

**Why this matters in C/C++**

Matrices represent linear transformations and systems of simultaneous equations. C++ numerical libraries use them everywhere from graphics to simulation and machine learning.

## Core ideas

- Matrix-vector multiplication combines input components into output components.
- Matrix multiplication composes linear transformations and is not generally commutative.
- The identity matrix leaves a vector unchanged.
- A matrix inverse exists only for a nonsingular square matrix, but explicitly forming an inverse is often unnecessary.
- Rank measures independent directions or constraints.
- Factorizations such as LU, QR, and Cholesky solve systems more efficiently and stably than ad-hoc elimination.

**Core mathematical tools**

$$Ax = b$$

$$(AB)x = A(Bx)$$

$$AI = IA = A$$

$\det(A) = 0 \Rightarrow A$ is singular for square $A$

**Where you will use it**

- linear solvers
- graphics transforms
- least squares
- physics constraints
- machine learning
- computer vision

CHAPTER 35

# Matrices, Linear Systems, Rank, and Decompositions - Engineering Lab

**Priority 9/10**

## Worked example

Solving Ax=b asks for variables x that satisfy several linear constraints simultaneously. In engineering code, using a tested decomposition from a numerical library is usually better than computing inv(A)*b.

## C++ connection

```cpp
#include <array>

using Mat2 = std::array<std::array<double,2>,2>;
using V2 = std::array<double,2>;

V2 mul(const Mat2& A, V2 x) {
   return {A[0][0]*x[0] + A[0][1]*x[1],
           A[1][0]*x[0] + A[1][1]*x[1]};
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** linear solvers, graphics transforms, least squares, physics constraints, machine learning, computer vision.

CHAPTER 35

# Matrices, Linear Systems, Rank, and Decompositions - Reliability Notes

**Priority 9/10**

**Failure modes**

- Assuming AB equals BA.
- Computing a matrix inverse when a solve is enough.
- Ignoring conditioning of a nearly singular system.
- Using row-major/column-major storage assumptions incorrectly across APIs.
- Forgetting dimensional compatibility in matrix products.

**Engineering checks**

- Write matrix dimensions beside formulas.
- Use library decompositions with documented numerical properties.
- Test identity and known-transform cases.
- Estimate residual ||Ax-b|| after a solve.

## Build intuition

For A=[[2,1],[1,3]] and x=[4,-1], compute Ax. Then solve the small system Ax=[7,1] by hand to connect matrix notation with simultaneous equations.

**Chapter takeaway**

Matrices are structured collections of linear relationships; dimensions and conditioning matter as much as the coefficients.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 36

# 2D/3D Transforms, Homogeneous Coordinates, and Coordinate Frames

**Priority 9/10**

**Why this matters in C/C++**

Transforms move data between local, world, camera, screen, robot, sensor, and model coordinate systems. Many difficult graphics and robotics bugs are simply frame-composition errors.

## Core ideas

- Translation is affine, not purely linear, so homogeneous coordinates embed it into matrix multiplication.
- Rotation preserves lengths and angles.
- Scaling changes length and may be non-uniform.
- Transform composition order matters.
- A point and a direction transform differently under translation.
- Changing basis expresses the same geometric object in a different coordinate frame.

**Core mathematical tools**

$$p' = Mp$$

$$T_{world \leftarrow local} = T_{world \leftarrow parent} T_{parent \leftarrow local}$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

**Where you will use it**

- rendering pipelines
- robot frames
- CAD
- game scene graphs
- computer vision
- UI transforms

CHAPTER 36

# 2D/3D Transforms, Homogeneous Coordinates, and Coordinate Frames - Engineering Lab

**Priority 9/10**

## Worked example

A local point first rotates in an object frame and then translates into world space. Reversing the matrix order translates the coordinate axes before rotating and produces a different result. The formula must state which frame each matrix maps from and to.

## C++ connection

```cpp
#include <array>

struct P2 { double x, y; };
P2 rotate(P2 p, double c, double s) {
   return {c*p.x - s*p.y, s*p.x + c*p.y};
}
P2 translate(P2 p, P2 t) { return {p.x+t.x, p.y+t.y}; }
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** rendering pipelines, robot frames, CAD, game scene graphs, computer vision, UI transforms.

CHAPTER 36

# 2D/3D Transforms, Homogeneous Coordinates, and Coordinate Frames - Reliability Notes

**Priority 9/10**

**Failure modes**

- Multiplying transforms in the wrong order.
- Mixing row-vector and column-vector conventions.
- Treating a normal exactly like a position under non-uniform scale.
- Mixing left-handed and right-handed coordinate systems.
- Applying translation to direction vectors.

**Engineering checks**

- Document matrix convention once per codebase.
- Name frames in transform variables.
- Test basis vectors and the origin.
- Use round-trip local->world->local tests.

## Build intuition

Take point (1,0). Rotate it 90 degrees around the origin, then translate by (10,2). Compare with translating first and rotating second.

**Chapter takeaway**

Coordinate-frame labels prevent more transform bugs than memorizing matrix entries.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 37

# Quaternions, Robust Geometry, and Collision Predicates

**Priority 9/10**

**Why this matters in C/C++**

Quaternions give compact 3D rotation composition and interpolation, while robust geometric predicates decide orientation, intersection, and collision. Together they support engines, CAD, robotics, and simulation.

## Core ideas

- A unit quaternion represents a 3D rotation.
- Quaternion multiplication composes rotations and is not commutative.
- q and -q represent the same physical rotation.
- SLERP interpolates along the unit quaternion sphere.
- Orientation tests use signed area/volume ideas.
- Robust predicates must account for floating error near degenerate geometry.

**Core mathematical tools**

$$q = (w, x, y, z), \quad \|q\| = 1$$

$$q^{-1} = \bar{q} \text{ for unit } q$$

$$\text{orient2d}(a, b, c) = (b_x - a_x)(c_y - a_y) - (b_y - a_y)(c_x - a_x)$$

**Where you will use it**

- camera orientation
- skeletal animation
- robot attitude
- collision detection
- CAD kernels
- computational geometry

CHAPTER 37

# Quaternions, Robust Geometry, and Collision Predicates - Engineering Lab

**Priority 9/10**

## Worked example

The sign of orient2d(a,b,c) tells whether c lies to the left or right of directed edge a->b. Near collinearity, rounding can flip the sign, so robust geometry libraries may use filtered or higher-precision predicates rather than a casual epsilon.

## C++ connection

```cpp
#include <cmath>

struct P2d { double x, y; };
double orient2d(P2d a, P2d b, P2d c) {
   return (b.x-a.x)*(c.y-a.y) -
          (b.y-a.y)*(c.x-a.x);
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** camera orientation, skeletal animation, robot attitude, collision detection, CAD kernels, computational geometry.

CHAPTER 37

# Quaternions, Robust Geometry, and Collision Predicates - Reliability Notes

**Priority 9/10**

**Failure modes**

- Interpolating quaternion components linearly without renormalizing or considering path choice.
- Forgetting q and -q equivalence in interpolation.
- Comparing orientation determinant to a universal epsilon.
- Using squared distance and distance interchangeably in threshold logic.
- Ignoring degenerate triangles, zero-area polygons, or touching collisions.

**Engineering checks**

- Normalize rotation inputs when drift is possible.
- Test identity and 180-degree-like edge cases.
- Use robust predicate libraries for topology-critical geometry.
- Separate broad-phase approximate collision from narrow-phase exact decisions.

## Build intuition

Compute orient2d for a=(0,0), b=(2,0), c=(1,3). What changes if c=(1,-3)? What should software do if the computed value is extremely close to zero?

**Chapter takeaway**

Spatial correctness depends on conventions and robust predicates as much as it depends on formulas.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

Part VII

# Calculus, Signals, Dynamics, and Control

Calculus matters most when software models change: motion, optimization, filters, control loops, physical systems, and continuous signals. The treatment focuses on intuition, discretization, and implementation rather than symbolic exercise drills.

CHAPTER 38

# Derivatives, Gradients, and Optimization

**Priority 9/10**

**Why this matters in C/C++**

A derivative measures local change. In software engineering it appears whenever a system must predict sensitivity, choose a step, optimize a parameter, estimate slope, or reason about motion.

## Core ideas

- The derivative is the local rate of change of a function.
- A gradient collects partial derivatives of a multivariable function.
- A local minimum often has zero gradient, but zero gradient does not guarantee a minimum.
- Finite differences approximate derivatives when an analytic derivative is unavailable.
- Gradient descent repeatedly moves opposite the gradient.
- Step size controls the trade between progress and stability.

**Core mathematical tools**

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

$$\nabla f = (\partial f / \partial x_1, \ldots)$$

$$x_{k+1} = x_k - \eta \nabla f(x_k)$$

**Where you will use it**

- optimization
- machine learning
- physics
- geometry fitting
- control tuning
- automatic differentiation

CHAPTER 38

# Derivatives, Gradients, and Optimization - Engineering Lab

**Priority 9/10**

## Worked example

For f(x)=(x-3)^2, the derivative is 2(x-3). Gradient descent updates x by subtracting eta*2(x-3), moving toward x=3 when eta is chosen sensibly. If eta is too large, the iteration can oscillate or diverge.

## C++ connection

```cpp
#include <cmath>

double minimize_quadratic(double x, double eta) {
   for (int i = 0; i < 100; ++i) {
      const double grad = 2.0 * (x - 3.0);
      x -= eta * grad;
   }
   return x;
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** optimization, machine learning, physics, geometry fitting, control tuning, automatic differentiation.

CHAPTER 38

# Derivatives, Gradients, and Optimization - Reliability Notes

**Priority 9/10**

**Failure modes**

- Using a finite-difference h that is too large or too small.
- Assuming every stationary point is a minimum.
- Choosing one step size for variables with radically different scales.
- Ignoring constraints on valid parameters.
- Stopping only because the iteration count ended.

**Engineering checks**

- Test derivatives against finite differences at ordinary points.
- Scale variables or use suitable preconditioning.
- Track objective decrease and gradient norm.
- Guard constraints and non-finite values.

## Build intuition

For f(x)=x^2+4x+10, derive the derivative, find its stationary point, and verify from the function shape that the point is a minimum.

**Chapter takeaway**

Derivatives quantify sensitivity; optimization is the disciplined use of that sensitivity to choose better parameters.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 39

# Integrals, Differential Equations, and Time Stepping

**Priority 9/10**

**Why this matters in C/C++**

An integral accumulates a rate; a differential equation describes how a state changes over time. Games, control, simulation, circuits, robotics, and physical models depend on turning these continuous ideas into discrete updates.

## Core ideas

- An integral accumulates infinitely small contributions and often represents area or total quantity.
- An ODE relates a state derivative to the current state, time, and inputs.
- Explicit Euler advances using the slope at the beginning of a step.
- Smaller steps usually reduce discretization error but increase cost.
- Stable time stepping depends on the dynamics, not only on floating precision.
- Fixed and variable time steps have different reproducibility and performance tradeoffs.

**Core mathematical tools**

$$x(t) = x(t_0) + \int_{t_0}^{t} v(\tau) d\tau$$

$$x_{n+1} = x_n + h f(t_n, x_n) \text{ (Euler)}$$

$$v_{n+1} = v_n + h a_n$$

**Where you will use it**

- game physics
- robotics
- circuit simulation
- control systems
- scientific models
- motion integration

CHAPTER 39

# Integrals, Differential Equations, and Time Stepping - Engineering Lab

**Priority 9/10**

## Worked example

With constant acceleration a=-9.81 m/s^2, explicit Euler updates velocity then position every h seconds. The result approaches the continuous solution as h shrinks, but energy and trajectory errors can accumulate. Method choice is part of simulation design.

## C++ connection

```
struct State { double x, v; };

State euler(State s, double a, double dt) {
   s.x += s.v * dt;
   s.v += a * dt;
   return s;
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** game physics, robotics, circuit simulation, control systems, scientific models, motion integration.

CHAPTER 39

# Integrals, Differential Equations, and Time Stepping - Reliability Notes

**Priority 9/10**

**Failure modes**

- Using frame duration as an unconstrained physics step.
- Assuming smaller dt fixes an unstable method in every system.
- Mixing seconds and milliseconds.
- Comparing simulation states with exact equality.
- Changing integration order and accidentally changing model behavior.

**Engineering checks**

- Validate against constant-velocity and constant-acceleration analytic cases.
- Track conserved quantities when the model should conserve them.
- Cap or subdivide extreme time steps.
- Separate display interpolation from simulation integration.

## Build intuition

A particle starts at x=0 with v=10 m/s and acceleration -2 m/s^2. Perform two Euler steps with dt=0.5 s and compare the position with the analytic constant-acceleration result.

**Chapter takeaway**

Continuous models become software through discretization; the time step and integrator are part of the mathematics.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 40

# Complex Numbers, Sampling, Convolution, and the FFT

**Priority 9/10**

**Why this matters in C/C++**

Complex numbers make oscillations and phase compact. Sampling turns continuous signals into sequences. Convolution and Fourier transforms connect time-domain behavior with frequency-domain structure.

## Core ideas

- A complex number has real and imaginary components and can also be represented by magnitude and phase.
- Euler's identity links complex exponentials to sine and cosine.
- Sampling must be fast enough for the signal bandwidth to avoid aliasing.
- Convolution combines a signal with an impulse response or kernel.
- The DFT decomposes a finite sequence into discrete frequency components.
- The FFT computes the DFT in O(n log n) instead of the direct O(n^2) work.

**Core mathematical tools**

$$e^{j\theta} = \cos\theta + j\sin\theta$$

$f_s > 2f_{max}$ is the basic Nyquist sampling condition

$$y[n] = \sum_{k} x[k]h[n-k]$$

$$X[k] = \sum_{n=0}^{N-1} x[n]e^{-j2\pi kn/N}$$

**Where you will use it**

- audio
- radio
- image filters
- spectral analysis
- communications
- vibration analysis

CHAPTER 40

# Complex Numbers, Sampling, Convolution, and the FFT - Engineering Lab

**Priority 9/10**

## Worked example

A 10 kHz sine sampled at 12 kHz cannot be represented unambiguously because the sample rate is below twice the signal frequency. It aliases to a lower apparent frequency. No later C++ filter can reconstruct information that was never sampled correctly.

## C++ connection

```cpp
#include <complex>
#include <numbers>

std::complex<double> phasor(double radians) {
   using cd = std::complex<double>;
   return std::exp(cd{0.0, radians});
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** audio, radio, image filters, spectral analysis, communications, vibration analysis.

CHAPTER 40

# Complex Numbers, Sampling, Convolution, and the FFT - Reliability Notes

**Priority 9/10**

**Failure modes**

- Believing sample rate alone defines usable bandwidth without an anti-alias filter.
- Using degrees in an API expecting radians.
- Ignoring FFT normalization conventions.
- Applying circular convolution when linear convolution is intended.
- Forgetting windowing and spectral leakage in finite measurements.

**Engineering checks**

- Label sample rates and frequencies in Hz.
- Test sinusoidal inputs at known FFT bins.
- Document FFT sign and normalization conventions.
- Use well-tested FFT libraries for production workloads.

## Build intuition

At a sample rate of 48 kHz, what is the Nyquist frequency? Why is a real-world anti-alias filter still required below that theoretical boundary?

**Chapter takeaway**

Sampling mathematics defines what information exists in the digital signal; algorithms can only work with what was captured.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 41

# Feedback Control, PID, State, and Discrete-Time Stability

**Priority 9/10**

**Why this matters in C/C++**

Embedded C/C++ often closes a loop around a physical process. Control mathematics turns sensor error into actuator commands while balancing speed, stability, noise, and saturation.

## Core ideas

- Feedback compares a measured output with a target and uses the error to adjust input.
- Proportional action responds to current error.
- Integral action accumulates past error and can remove steady-state bias.
- Derivative action responds to error change and is sensitive to noise.
- Discrete implementation introduces a sample period that must appear in integral and derivative terms.
- Actuator saturation creates windup unless the controller design handles it.

**Core mathematical tools**

$$e_k = r_k - y_k$$

$$u_k = K_p e_k + K_i \sum e_k \Delta t + K_d \frac{e_k - e_{k-1}}{\Delta t}$$

sample period $\Delta t$ is part of controller dynamics

**Where you will use it**

- motor control
- temperature control
- drones
- robotics
- power electronics
- industrial systems

CHAPTER 41

# Feedback Control, PID, State, and Discrete-Time Stability - Engineering Lab

**Priority 9/10**

## Worked example

A PID loop running every 1 ms cannot simply reuse coefficients tuned for a 10 ms implementation if the code stores an unscaled integral sum and finite-difference derivative. The discrete equations change with dt.

## C++ connection

```cpp
#include <algorithm>

struct PID {
   double kp{}, ki{}, kd{}, integral{}, prev{};
   double step(double error, double dt) {
      integral += error * dt;
      const double derivative = (error - prev) / dt;
      prev = error;
      return kp*error + ki*integral + kd*derivative;
   }
};
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** motor control, temperature control, drones, robotics, power electronics, industrial systems.

CHAPTER 41

# Feedback Control, PID, State, and Discrete-Time Stability - Reliability Notes

**Priority 9/10**

**Failure modes**

- Dividing by dt without validating it.
- Integral windup under saturated actuators.
- Differentiating noisy measurements without filtering or design care.
- Changing loop frequency without retuning or rescaling.
- Using average execution time in a deadline-sensitive controller.

**Engineering checks**

- Log setpoint, measurement, error, and command.
- Test saturation and recovery.
- Measure actual loop jitter.
- Use a simple plant model or hardware-in-the-loop before aggressive tuning.

## Build intuition

If a PI controller has Kp=2, Ki=0.5, dt=0.1 s, and constant error 3 for four steps, compute the proportional and accumulated integral contributions after the fourth step.

**Chapter takeaway**

A control loop is mathematics executing on a clock. Timing, saturation, units, and noise belong in the controller model.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

Part VIII

# Mathematics by C/C++ Specialization

The earlier chapters build the common core. This part shows how those ideas combine inside major C and C++ specialties. Each chapter is a map: it tells an engineer which mathematics becomes critical, why, and what implementation traps are specific to the domain.

Progression: practical foundations → rigorous reasoning → machine reliability → specialist mathematics

CHAPTER 42

# Graphics, Rendering, Color, and Imaging

**Priority 9/10**

**Why this matters in C/C++**

Graphics combines geometry, linear algebra, probability, sampling, and numerical precision. Even a small renderer moves repeatedly between coordinate spaces, projected areas, normalized vectors, colors, and sampled light.

## Core ideas

- Perspective projection uses ratios and homogeneous coordinates.
- Normals and dot products control many lighting terms.
- Color values may live in non-linear transfer functions; arithmetic should be performed in the intended color space.
- Texture filtering is interpolation plus sampling theory.
- Monte Carlo rendering estimates integrals from random samples.
- Depth-buffer precision is non-linear under common perspective mappings.

**Core mathematical tools**

$$x_{screen} \propto \frac{x_{camera}}{z_{camera}}$$

$$L_o \approx \frac{1}{N} \sum_{i=1}^{N} \frac{f(X_i)}{p(X_i)}$$

$$\text{Lambert term} = \max(0, n \cdot l)$$

**Where you will use it**

- rasterizers
- ray tracers
- image pipelines
- GPU kernels
- color management
- camera simulation

CHAPTER 42

# Graphics, Rendering, Color, and Imaging - Engineering Lab

**Priority 9/10**

## Worked example

A normalized surface normal n and light direction l produce diffuse intensity proportional to max(0,n dot l). If either vector is not normalized, the result also contains unwanted length scale. One missing normalization can look like a mysterious brightness bug.

## C++ connection

```cpp
#include <algorithm>

double lambert(Vec3 unit_n, Vec3 unit_l) {
   return std::max(0.0, dot(unit_n, unit_l));
}

// Color-space conversions and transfer functions should be explicit;
// do not assume stored RGB values are linear-light intensities.
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** rasterizers, ray tracers, image pipelines, GPU kernels, color management, camera simulation.

CHAPTER 42

# Graphics, Rendering, Color, and Imaging - Reliability Notes

**Priority 9/10**

**Failure modes**

- Blending non-linear encoded colors as though they were linear light.
- Using degrees/radians inconsistently.
- Ignoring perspective divide or w=0-like degeneracy.
- Normalizing zero-length geometry.
- Treating random-sample count as the only source of rendering error.

**Engineering checks**

- Render canonical test scenes with known normals and colors.
- Test coordinate transforms and their inverses.
- Use image-difference metrics appropriate to the task.
- Separate geometric precision problems from sampling noise.

## Build intuition

A surface normal forms 60 degrees with a normalized light direction. Compute the Lambert diffuse factor. What happens at 100 degrees?

**Chapter takeaway**

Graphics is where linear algebra, geometry, sampling, and probability become visible immediately on screen.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 43

# Game Physics, Collision, and Simulation

**Priority 9/10**

**Why this matters in C/C++**

Game and interactive simulation code mixes vectors, integration, collision geometry, impulses, interpolation, and probabilistic systems under a strict frame-time budget.

## Core ideas

- Position, velocity, acceleration, force, and impulse have different units.
- Collision detection often separates broad-phase spatial pruning from narrow-phase geometric tests.
- Impulse methods change momentum over short events.
- Constraint solvers iteratively reduce violations rather than solving an exact symbolic system every frame.
- Fixed simulation steps improve consistency; rendering can interpolate between simulation states.
- Energy drift reveals numerical properties of the integrator.

**Core mathematical tools**

$$F = ma$$

$$p = mv$$

$$J = \Delta p$$

$$x_{n+1} = x_n + v_n \Delta t$$

**Where you will use it**

- game engines
- VR
- interactive simulation
- particle systems
- collision engines

CHAPTER 43

# Game Physics, Collision, and Simulation - Engineering Lab

**Priority 9/10**

## Worked example

A 2 kg body moving at 5 m/s has momentum 10 kg m/s. An impulse of -4 N s along the same axis changes momentum to 6 and velocity to 3 m/s. Keeping force, impulse, and velocity units distinct prevents many “physics tuning” mistakes.

## C++ connection

```
struct Body1D { double mass, velocity; };

void apply_impulse(Body1D& b, double impulse) {
   if (b.mass > 0.0) b.velocity += impulse / b.mass;
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** game engines, VR, interactive simulation, particle systems, collision engines.

CHAPTER 43

# Game Physics, Collision, and Simulation - Reliability Notes

**Priority 9/10**

**Failure modes**

- Applying frame-rate-dependent forces without dt.
- Correcting penetration with huge impulses that destabilize the simulation.
- Mixing world and local coordinates.
- Using exact equality for contact geometry.
- Allowing one long frame to become one enormous physics step.

**Engineering checks**

- Check units on every state variable.
- Use deterministic regression scenes for core solver behavior.
- Monitor energy/momentum where the model predicts conservation.
- Test slow, fast, grazing, touching, and stacked contacts.

## Build intuition

A 4 kg body at 2 m/s collides head-on with a stationary 1 kg body. Compute total momentum before collision and explain which additional physical assumption is needed to determine both final velocities.

**Chapter takeaway**

Game physics is numerical modeling under a time budget; stability and units matter more than visually plausible guesses.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 44

# Security, Cryptography, CRCs, and Information Theory

**Priority 10/10**

**Why this matters in C/C++**

Security-oriented C/C++ uses modular arithmetic, finite fields, probability, entropy, bit operations, and strict range reasoning. The mathematics is essential, but secure implementation also requires side-channel awareness and reviewed algorithms.

## Core ideas

- Cryptographic modular arithmetic works over precisely defined integer structures.
- Entropy measures uncertainty, not simply string length.
- Cryptographic hash security depends on preimage and collision properties, not ordinary hash-table behavior.
- CRCs use polynomial arithmetic over GF(2) for error detection, not secrecy.
- Constant-time intent concerns information leakage through execution behavior, not mathematical correctness alone.
- Random keys require a cryptographically secure random source and sufficient entropy.

**Core mathematical tools**

$$H(X) = -\sum_{x} p(x) \log_2 p(x)$$

$$a^e \bmod m$$

CRC arithmetic uses coefficients modulo $2$

**Where you will use it**

- TLS and protocol libraries
- storage integrity
- firmware
- authentication
- secure parsers
- error-detecting codes

CHAPTER 44

# Security, Cryptography, CRCs, and Information Theory - Engineering Lab

**Priority 10/10**

## Worked example

XOR in CRC arithmetic corresponds to coefficient addition modulo 2. There are no carries. This is why CRC implementations look like shifts and XORs even though the underlying model is polynomial division. That mathematics detects common transmission errors but does not provide authentication.

## C++ connection

```cpp
#include <cstdint>

std::uint32_t parity32(std::uint32_t x) {
   x ^= x >> 16; x ^= x >> 8; x ^= x >> 4;
   x &= 0xF;
   return (0x6996u >> x) & 1u;
}

// Educational bit arithmetic only. Use reviewed crypto libraries
// for encryption, authentication, signatures, and key handling.
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** TLS and protocol libraries, storage integrity, firmware, authentication, secure parsers, error-detecting codes.

CHAPTER 44

# Security, Cryptography, CRCs, and Information Theory - Reliability Notes

**Priority 10/10**

**Failure modes**

- Implementing custom production cryptography from textbook snippets.
- Using rand() for keys or security tokens.
- Treating CRC as a MAC.
- Leaking secrets through table lookups, branches, or timing.
- Using ordinary integer overflow where a cryptographic specification requires exact modular operations.

**Engineering checks**

- Use established libraries and protocols.
- Validate every length before allocation or copy.
- Use test vectors from the algorithm specification.
- Review randomness, side channels, key lifecycle, and error handling separately.

## Build intuition

A source has four equally likely symbols. Compute its entropy in bits per symbol. Then explain why a human-chosen four-character password does not automatically have the same entropy model.

**Chapter takeaway**

Security mathematics is necessary but not sufficient: use correct primitives, correct implementations, and explicit threat models.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 45

# Networking, Storage, Databases, and Distributed Systems

**Priority 9/10**

**Why this matters in C/C++**

Systems that move or persist data rely on rates, queueing, checksums, probability, replication math, timeouts, ordering, and capacity calculations. C/C++ often implements the performance-sensitive layers where these details matter most.

## Core ideas

- Serialization size is a sum of field sizes plus framing and alignment overhead.
- Bandwidth-delay product estimates data that can be in flight.
- Queueing grows sharply as offered load approaches service capacity.
- Replication increases durability/availability under assumptions about independent failures.
- Checksums detect corruption with defined error models.
- Timeout and retry policies create feedback and can amplify load.

**Core mathematical tools**

$$\text{BDP} = \text{bandwidth} \times \text{round-trip time}$$

$$\text{capacity} = \text{records} \times \text{bytes/record}$$

$$L = \lambda W$$

**Where you will use it**

- network stacks
- database engines
- storage systems
- RPC frameworks
- distributed caches
- message brokers

CHAPTER 45

# Networking, Storage, Databases, and Distributed Systems - Engineering Lab

**Priority 9/10**

## Worked example

On a 10 Gbit/s path with 40 ms round-trip time, the bandwidth-delay product is about 50 MB in decimal units. A sender needs enough in-flight data to fill that pipeline; a tiny window leaves bandwidth unused even when each packet is fast.

## C++ connection

```cpp
#include <cstdint>

std::uint64_t bandwidth_delay_bytes(
   std::uint64_t bits_per_second, double rtt_seconds) {
   return static_cast<std::uint64_t>(
      (double(bits_per_second) * rtt_seconds) / 8.0);
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** network stacks, database engines, storage systems, RPC frameworks, distributed caches, message brokers.

CHAPTER 45

# Networking, Storage, Databases, and Distributed Systems - Reliability Notes

**Priority 9/10**

**Failure modes**

- Confusing bits/s and bytes/s.
- Ignoring protocol and framing overhead.
- Assuming node failures are independent when they share power/rack/software.
- Using wall-clock time as a monotonic duration source.
- Retrying immediately and synchronously during overload.

**Engineering checks**

- Carry units through capacity calculations.
- Use steady/monotonic clocks for elapsed-time logic.
- Model failure correlation explicitly.
- Load-test retry and backoff behavior under partial failure.

## Build intuition

A record averages 640 bytes including indexes and metadata. Estimate raw storage for 250 million records, then multiply by a replication factor of 3 before filesystem or compression effects.

**Chapter takeaway**

Distributed systems turn simple arithmetic into operational limits; rates, queues, time, and correlated failures must be kept explicit.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 46

# Compilers, Parsers, Data-Flow Lattices, and Program Analysis

**Priority 9/10**

**Why this matters in C/C++**

Compiler C/C++ uses discrete mathematics heavily: automata, grammars, graphs, sets, fixed points, dominance, bitsets, and cost models. The mathematics makes analyses composable and proves when iteration can stop.

## Core ideas

- Lexers are often modeled by finite automata; parsers use grammars and stack-like structures.
- Control-flow graphs model possible execution transfers.
- Dominance is a relation on CFG nodes.
- Many data-flow analyses operate over a lattice of facts.
- Monotone transfer functions and finite-height domains allow fixed-point iteration to converge.
- Bitsets implement finite set operations efficiently.

**Core mathematical tools**

$$IN[B] = \bigwedge_{P \in pred(B)} OUT[P]$$

$$OUT[B] = F_B(IN[B])$$

$$x_{k+1} = F(x_k) \text{ until } x_{k+1} = x_k$$

**Where you will use it**

- optimizing compilers
- static analyzers
- assemblers
- JITs
- link-time analysis
- verification tools

CHAPTER 46

# Compilers, Parsers, Data-Flow Lattices, and Program Analysis - Engineering Lab

**Priority 9/10**

## Worked example

Live-variable analysis repeatedly propagates sets backward through a CFG until no set changes. Union, set difference, and equality are the core operations. The fixed-point perspective explains why a worklist is more natural than a fixed number of arbitrary passes.

## C++ connection

```cpp
#include <bitset>

using Facts = std::bitset<256>;
Facts transfer(Facts live_out, Facts def, Facts use) {
   return use | (live_out & ~def);
}
// A worklist repeatedly applies transfer until facts stop changing.
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** optimizing compilers, static analyzers, assemblers, JITs, link-time analysis, verification tools.

CHAPTER 46

# Compilers, Parsers, Data-Flow Lattices, and Program Analysis - Reliability Notes

**Priority 9/10**

**Failure modes**

- Iterating a data-flow pass a guessed number of times.
- Using a meet operation that does not match the analysis meaning.
- Confusing dominance with reachability.
- Allowing an analysis domain to grow without convergence guarantees.
- Treating parser complexity independently from grammar shape and ambiguity.

**Engineering checks**

- Test analyses on tiny CFGs where facts are computable by hand.
- Assert monotone progress where expected.
- Track worklist iterations and domain changes.
- Use graph visualization for control-flow debugging.

## Build intuition

For live variables, explain why a use of x makes x live before the instruction, while a definition of x can kill the previous value. Express this as set operations.

**Chapter takeaway**

Compiler mathematics turns “repeat until it seems done” into a fixed-point computation with explicit structure and convergence reasoning.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 47

# HPC, Scientific Computing, and Machine Learning Kernels

**Priority 10/10**

**Why this matters in C/C++**

High-performance C++ combines linear algebra, numerical analysis, probability, parallelism, SIMD, memory models, and performance bounds. The fastest code is often the algorithm that moves less data and maintains useful numerical structure.

## Core ideas

- Dense linear algebra is dominated by structured matrix/vector operations.
- Operational intensity helps classify kernels as compute-bound or bandwidth-bound.
- Parallel reductions change operation order and floating results.
- Condition numbers and solver convergence determine whether more FLOPs improve accuracy.
- ML kernels frequently reduce to matrix multiplication, convolution, normalization, and non-linear functions.
- Mixed precision trades range/accuracy against bandwidth and throughput.

**Core mathematical tools**

$$\text{AI} = \frac{\text{FLOPs}}{\text{bytes moved}}$$

$$\text{attainable performance} \lesssim \min(P_{peak}, B \times AI)$$

$\|Ax - b\|$ is a useful solver residual

**Where you will use it**

- simulation
- BLAS-like kernels
- AI inference/training components
- weather/science codes
- computational finance
- large-scale analytics

CHAPTER 47

# HPC, Scientific Computing, and Machine Learning Kernels - Engineering Lab

**Priority 10/10**

## Worked example

A vector-add kernel performs one floating add while reading two inputs and writing one output. Its arithmetic intensity is extremely low, so widening SIMD may not help once memory bandwidth is saturated. A matrix-multiply kernel reuses data and can have much higher intensity.

## C++ connection

```cpp
#include <span>

void axpy(double a, std::span<const double> x,
          std::span<double> y) {
   for (std::size_t i = 0; i < x.size(); ++i)
      y[i] = a * x[i] + y[i];
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** simulation, BLAS-like kernels, AI inference/training components, weather/science codes, computational finance, large-scale analytics.

CHAPTER 47

# HPC, Scientific Computing, and Machine Learning Kernels - Reliability Notes

**Priority 10/10**

**Failure modes**

- Reporting peak FLOPs as application performance.
- Ignoring residual/error while optimizing solver time.
- Changing reduction order and expecting exact reproducibility.
- Using lower precision without checking accumulated error and range.
- Benchmarking data that fits in cache when production data does not.

**Engineering checks**

- Build a roofline-style bytes/FLOPs estimate first.
- Validate against a higher-precision or reference implementation.
- Benchmark multiple working-set sizes.
- Report numerical error together with time-to-solution.

## Build intuition

For y=a*x+y using doubles, count approximate bytes read/written and floating operations per element. Estimate arithmetic intensity before considering cache write policies.

**Chapter takeaway**

HPC performance is constrained by both mathematics and hardware: accuracy, bytes, operations, parallelism, and convergence all matter.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 48

# Embedded, Robotics, Sensor Fusion, and Geospatial Computing

**Priority 10/10**

**Why this matters in C/C++**

Embedded and robotics software connects integer representation, fixed point, real-time timing, coordinate transforms, control, probability, and sensor error. Geospatial work adds spherical/ellipsoidal coordinates and severe scale differences.

## Core ideas

- Sensors measure noisy physical quantities with units and calibration models.
- Coordinate frames must be named and transformed consistently.
- Fusion combines uncertain measurements rather than simply averaging all sensors equally.
- Covariance describes uncertainty and correlation in state estimates.
- Real-time loops require bounded execution and stable sample periods.
- Latitude/longitude are angular coordinates; flat-Earth approximations fail over large distances or near special regions.

**Core mathematical tools**

$$z = Hx + v$$

$K$ weights prediction versus measurement in Kalman-style estimation

$d \approx R\Delta\theta$ for small angular displacement

**Where you will use it**

- autonomous systems
- drones
- industrial control
- IoT
- GNSS/GPS processing
- embedded navigation

CHAPTER 48

# Embedded, Robotics, Sensor Fusion, and Geospatial Computing - Engineering Lab

**Priority 10/10**

## Worked example

Two sensors with very different noise should not receive equal trust merely because both output the same unit. A fusion algorithm weights information according to an uncertainty model, then transforms all measurements into compatible coordinate frames.

## C++ connection

```cpp
struct Measurement { double value, variance; };

double fuse_two(Measurement a, Measurement b) {
   const double wa = 1.0 / a.variance;
   const double wb = 1.0 / b.variance;
   return (wa*a.value + wb*b.value) / (wa + wb);
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** autonomous systems, drones, industrial control, IoT, GNSS/GPS processing, embedded navigation.

CHAPTER 48

# Embedded, Robotics, Sensor Fusion, and Geospatial Computing - Reliability Notes

**Priority 10/10**

**Failure modes**

- Mixing degrees and radians.
- Averaging angles across the -pi/pi wrap as ordinary scalars.
- Treating sensor errors as independent when they share a bias.
- Using desktop-style unbounded allocation inside hard real-time loops.
- Computing global distances with a local planar approximation without an error budget.

**Engineering checks**

- Annotate every signal with unit, frame, timestamp, and uncertainty.
- Test wrap-around angles and coordinate singularities.
- Simulate bias, noise, dropout, and delay.
- Measure worst-case loop execution, not only average.

## Build intuition

Two unbiased scalar sensors have variances 1 and 9. Compute inverse-variance weights and explain why the lower-variance sensor receives more weight.

**Chapter takeaway**

Robotics software becomes tractable when every number carries four meanings: unit, frame, time, and uncertainty.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

CHAPTER 49

# Finance, Time, Decimal Quantities, and Deterministic Business Math

**Priority 9/10**

**Why this matters in C/C++**

C++ also powers exchanges, pricing engines, ledgers, and low-latency business systems. Here the key mathematics is often exact discrete arithmetic, rates, compounding, rounding policy, statistics, and time rather than geometry.

## Core ideas

- Money often needs decimal or scaled-integer semantics rather than binary floating approximation.
- Rounding mode is a business rule, not an implementation detail.
- Percentages and rates require a clear base and period.
- Compound growth multiplies factors over periods.
- Time points and durations are different quantities.
- Financial risk and latency models require distributions and tail statistics.

**Core mathematical tools**

$$A = P(1 + r)^n$$

$$\text{return} = \frac{V_1 - V_0}{V_0}$$

$$\text{basis points} = 10^{-4} \text{ of the base quantity}$$

**Where you will use it**

- trading systems
- billing
- ledgers
- pricing
- risk analytics
- low-latency market data

CHAPTER 49

# Finance, Time, Decimal Quantities, and Deterministic Business Math - Engineering Lab

**Priority 9/10**

## Worked example

Representing cents as an integer can make addition exact for a currency with two decimal places, but multiplication by tax or interest still requires a defined rounding step. The correct rounded result depends on business and regulatory rules, not on a generic epsilon.

## C++ connection

```cpp
#include <cstdint>

using Cents = std::int64_t;

Cents add_money(Cents a, Cents b) {
   // In production, use checked addition and a domain-specific money type.
   return a + b;
}
```

## What to notice

- The mathematics is written before the optimization. Make the quantities and assumptions explicit first.
- The C++ type and evaluation rules are part of the computation, especially for sizes, integer ranges, and floating-point expressions.
- The implementation should expose preconditions and edge cases rather than depending on accidental behavior.
- Specialized libraries can improve performance and reliability, but you still need this mathematics to select, configure, and validate them.

**Where you will use it**

**Specialty connections:** trading systems, billing, ledgers, pricing, risk analytics, low-latency market data.

CHAPTER 49

# Finance, Time, Decimal Quantities, and Deterministic Business Math - Reliability Notes

**Priority 9/10**

**Failure modes**

- Using double for ledger equality.
- Rounding at inconsistent stages.
- Mixing annual, monthly, and per-second rates.
- Using system_clock duration differences where a monotonic clock is needed.
- Ignoring overflow in scaled integer amounts.

**Engineering checks**

- Define currency scale and rounding policy centrally.
- Use checked integer arithmetic for stored amounts.
- Test halfway rounding cases explicitly.
- Keep time zones/calendar logic separate from monotonic latency measurement.

## Build intuition

A balance of 1000 grows by 1 percent per month for 12 months. Compare simple 12 percent growth with monthly compounding and explain why the results differ.

**Chapter takeaway**

Business mathematics is often simple but unforgiving: exact units, scales, periods, and rounding rules are the engineering challenge.

## Before you move on

- Can you explain the idea without using a formula first?
- Can you identify the units, valid domain, and representable range?
- Can you name at least one C/C++ failure mode that the pure mathematical formula does not show?
- Can you construct a tiny input whose answer you can verify by hand?

Reference Section

# The C/C++ Engineer’s Math Toolkit

A compact set of formulas, library maps, test strategies, exercises, and specialization roadmaps for daily use.

Progression: practical foundations → rigorous reasoning → machine reliability → specialist mathematics

# Standard-Library Mathematics Map

| Header / Facility | Engineering use |
|---|---|
| `<cmath>` | Elementary functions, roots, powers, logs, trigonometry, classification, `hypot`, `fma`, `lerp`, `log1p`, `expm1`, and rounding operations. |
| `<numbers>` | Named mathematical constants such as $\pi$ and $e$ with type-aware templates in modern C++. |
| `<complex>` | Complex arithmetic for DSP, FFT-facing code, waves, controls, and scientific work. |
| `<numeric>` | Accumulation, inner products, partial sums, gcd/lcm, midpoint, and numerical algorithms. |
| `<bit>` | Bit width, population count, rotations, endian information, and power-of-two helpers. |
| `<random>` | Engines and probability distributions for simulation and non-security randomized algorithms. |
| `<chrono>` | Typed durations and time points; essential for unit-safe timing and latency measurement. |
| `<limits>` | Type ranges, precision characteristics, infinities/NaNs when supported, and implementation properties. |
| `<ratio>` | Compile-time rational ratios used by facilities such as chrono and custom quantity systems. |
| C headers | `math.h`, `stdint.h`, `limits.h`, `float.h`, `fenv.h`, and implementation-specific SIMD/intrinsic interfaces when appropriate. |

**Failure modes**

The standard library provides building blocks, not a full linear-algebra, arbitrary-precision, FFT, computer-vision, or cryptographic stack. Choose domain libraries according to accuracy, performance, licensing, portability, and validation requirements.

# Safe Arithmetic Recipes

## Checked size multiplication

Check $b \leq \max / a$ before computing $a \times b$ when $a \neq 0$.

## Midpoints

Prefer `std::midpoint` when the midpoint itself is the intent; the naive expression $(a+b)/2$ can overflow for integer endpoints.

## Range mapping

Normalize the source interval, then interpolate the destination interval. Validate degenerate source ranges.

## Floating comparison

Use a tolerance based on both absolute scale and relative magnitude. Domain requirements define the tolerance; machine epsilon alone does not.

## Stable elementary functions

Use specialized functions such as `log1p(x)` for $\log(1 + x)$ near zero and `expm1(x)` for $e^x - 1$ near zero when they fit the problem.

## Accumulators

Use a wider integer accumulator when totals can exceed element range. For floating reductions, consider pairwise or compensated summation when error matters.

# Choosing the Mathematics by Specialization

| Specialization | Highest-priority mathematics |
|---|---|
| **Systems / OS** | integer ranges, modular arithmetic, alignment, graphs, probability, queueing, performance models, concurrency order |
| **Embedded / real time** | fixed point, units, control, sampling, timing, modular counters, worst-case analysis |
| **Graphics / games** | vectors, matrices, transforms, geometry, calculus, probability, sampling, numerical stability |
| **HPC / science** | linear algebra, floating point, conditioning, error, calculus, probability, SIMD, roofline/performance models |
| **Networking / storage** | rates, queueing, probability, CRCs, modular arithmetic, time, capacity, failure models |
| **Compilers / tools** | logic, sets, graphs, lattices, fixed points, combinatorics, asymptotics, bitsets |
| **Security** | modular arithmetic, number theory, finite fields, probability, entropy, exact integer reasoning |
| **DSP / audio / image** | complex numbers, sampling, convolution, FFT, fixed/floating point, filters, statistics |
| **Robotics / control** | vectors, transforms, quaternions, ODEs, control, probability/covariance, timing |
| **Finance / low latency** | exact scaled arithmetic, rates, compounding, statistics, time, queueing, integer safety |

# Formula Atlas - Foundations

**Affine mapping**
$y = y_0 + (x - x_0)\frac{y_1 - y_0}{x_1 - x_0}$

**Linear interpolation**
$\mathsf{lerp}(a, b, t) = a + t(b - a)$

**Geometric sum**
$\sum_{i=0}^{n-1} ar^i = a(1 - r^n)/(1 - r)$

**Triangular sum**
$\sum_{i=1}^{n} i = n(n + 1)/2$

**Combinations**
$\binom{n}{r} = n!/[r!(n - r)!]$

**Conditional probability**
$P(A|B) = P(A \cap B)/P(B)$

**Bayes**
$P(A|B) = P(B|A)P(A)/P(B)$

**Expectation**
$E[X] = \sum_x xP(X = x)$

**Variance**
$Var(X) = E[X^2] - E[X]^2$

**Amdahl**
$S = 1/[(1 - p) + p/N]$

**Little**
$L = \lambda W$

**Utilization**
$U = \sum_i C_i/T_i$

# Formula Atlas - Vectors, Numerical Work, and Signals

**Vector norm**
$\|v\| = \sqrt{v \cdot v}$

**Dot product**
$a \cdot b = \|a\|\|b\| \cos\theta$

**Projection**
$proj_b(a) = (a \cdot b)b/(b \cdot b)$

**Euler step**
$x_{n+1} = x_n + hf(t_n, x_n)$

**Newton iteration**
$x_{n+1} = x_n - f(x_n)/f'(x_n)$

**Relative error**
$|\hat{x} - x|/|x|$

**DFT**
$X[k] = \sum_{n=0}^{N-1} x[n]e^{-j2\pi kn/N}$

**Convolution**
$y[n] = \sum_k x[k]h[n-k]$

**Entropy**
$H(X) = -\sum_x p(x) \log_2 p(x)$

**Bandwidth-delay product**
$BDP = bandwidth \times RTT$

**Arithmetic intensity**
$AI = operations/bytes$

# 60 Micro-Exercises

1. Convert 750 MiB to bytes.
2. Map 25 from [0,100] to [-1,1].
3. Find the next power of two at least 5000.
4. Write 0xA7 in binary.
5. Simplify $\neg(A \wedge B)$.
6. Compute $|A \cup B|$ when sizes are 20, 15, overlap 4.
7. Sum integers 1 through 1000.
8. Count 5-element combinations from 12.
9. State a loop invariant for array minimum.
10. Estimate depth of balanced binary tree with one million nodes.
11. Compute gcd(462,1071).
12. Compute lcm(12,18,30).
13. Find 1000 mod 64.
14. Find probability of at least one success in 10 independent trials with p=.1.
15. Compute expected cost for outcomes 3 with p=.8 and 20 with p=.2.
16. Explain mean versus median for one huge outlier.
17. Explain why correlation is not causation.
18. Estimate standard error if s=10 and n=100.
19. Compute Bloom optimal k approximately for 10 bits/key.
20. Give max value of uint16_t.
21. Explain why signed overflow cannot be used for portable wrap.
22. Convert raw Q8.8 value 384 to real value.
23. Explain why 0.1 is not exact in binary floating point.
24. Give a scale-aware floating comparison rule.
25. Name one cause of catastrophic cancellation.
26. Estimate output error using a derivative.
27. Count bisection iterations for interval reduction by $2^{20}$.
28. Align byte offset 1000 up to 64 bytes.
29. Compute row-major index for row 7, column 3, 20 columns.
30. Compute lower-bound scan time for 5 GiB at 50 GiB/s.
31. Compute lanes in a 256-bit vector of doubles.
32. Apply Amdahl with p=.9 and N=8.
33. Apply Little with 1000 items/s and 0.02 s average time.
34. Compute real-time utilization for (1,5),(2,20).
35. Find norm of (3,4).
36. Find dot product of (1,2,3) and (4,0,-1).
37. Project (3,4) onto x-axis.
38. Multiply a 2x2 matrix by a vector.
39. Explain why transform order matters.
40. Evaluate orient2d on one left-turn triangle.
41. Differentiate $(x-4)^2$.
42. Perform one Euler step for constant velocity.
43. State Nyquist frequency for 44.1 kHz sampling.
44. Explain convolution in one sentence.
45. Compute PID proportional term for Kp=2, error=3.
46. Compute Lambert term for angle 60 degrees.
47. Compute momentum for 3 kg at 4 m/s.
48. Compute entropy of eight equiprobable symbols.
49. Compute BDP for 1 Gbit/s and 20 ms.
50. Express live-variable transfer with use, def, live-out sets.
51. Compute arithmetic intensity of one add with two reads and one write of floats.
52. Fuse variances 1 and 4 using inverse-variance weighting.
53. Compare simple and compounded 1 percent monthly growth.
54. Explain false sharing in one sentence.
55. State difference between latency and throughput.
56. State difference between conditioning and stability.
57. State one reason a Poisson model can fail.
58. Explain why CRC is not authentication.
59. Name four meanings that should accompany a robot sensor number.
60. Choose the three most important math areas for your own C/C++ specialty and justify them.

# Answer Sketches

The exercises are designed for hand calculation and explanation. Selected numerical checks: 1) $750 \times 2^{20} = 786{,}432{,}000$ bytes. 2) $-0.5$. 3) 8192. 4) 1010 0111. 6) 31. 7) 500500. 11) 21. 12) 180. 13) 40. 15) 6.4. 18) 1 under the standard $s/\sqrt{n}$ estimate. 20) 65535. 22) 1.5. 27) 20 bisections give a $2^{20}$ reduction. 28) 1024. 29) 143 elements from zero-based indexing. 30) 0.1 s lower bound. 31) 4 lanes. 33) $L = 20$. 34) 0.3. 35) 5. 36) 1. 41) derivative $2(x-4)$. 43) 22.05 kHz. 45) 6. 47) 12 kg m/s. 48) 3 bits/symbol. Remaining items are intentionally explanatory; compare your answer with the relevant chapter's reliability page.

**Best way to use the exercises**

Do not chase arithmetic speed. For each answer, also state the type, unit, valid range, and one implementation failure mode. That extra sentence is what turns mathematics practice into C/C++ engineering practice.

# Glossary

| Term | Plain-English meaning |
|---|---|
| **Absolute error** | The magnitude of the difference between an approximation and a reference value. |
| **Alignment** | A divisibility requirement on an address or offset. |
| **Amortized cost** | Average cost per operation across a sequence, even when occasional operations are expensive. |
| **Conditioning** | Sensitivity of a mathematical problem to small input changes. |
| **Covariance** | A measure of how two variables vary together. |
| **Entropy** | A measure of uncertainty/information in a probability model. |
| **Gradient** | Vector of partial derivatives of a scalar function. |
| **Invariant** | A property intended to remain true at a specific program point or state. |
| **Latency** | Time required for one operation or request to complete. |
| **Modulo** | Arithmetic on remainder classes, useful for cyclic state. |
| **Norm** | A function measuring vector magnitude. |
| **Percentile** | A threshold at or below which a chosen percentage of samples lies. |
| **Projection** | Component of one vector along another. |
| **Quantization** | Mapping values to a finite set of representable levels. |
| **Recurrence** | A definition or cost relation expressed using smaller instances. |
| **Residual** | Difference that remains when a computed solution is substituted into an equation, such as $Ax - b$. |
| **Stability** | How an algorithm controls numerical error during computation. |
| **Throughput** | Completed work per unit time. |
| **Variance** | Expected squared spread around the mean. |
| **Working set** | Data actively needed over an interval of execution. |

# Selected References and Next Steps

- C and C++ language references and the documentation of the exact compiler and standard library used by the project.
- David Goldberg, *What Every Computer Scientist Should Know About Floating-Point Arithmetic*.
- Nicholas J. Higham, *Accuracy and Stability of Numerical Algorithms*.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*.
- Donald E. Knuth, *The Art of Computer Programming*, especially the volumes relevant to algorithms and seminumerical methods.
- Numerical linear-algebra documentation from established libraries used in your ecosystem, together with texts on matrix computation and numerical methods.
- Architecture manuals and performance guides from the target processor vendor when working with SIMD, caches, counters, or low-level performance.
- Reviewed cryptographic library and protocol documentation for all security-sensitive work; do not treat a general mathematics text as an implementation specification.

**The final idea**

A C or C++ engineer does not need to become a pure mathematician. The goal is to become mathematically fluent enough that quantities, limits, uncertainty, numerical representation, geometry, performance, and correctness are visible while reading code. When the mathematics is visible, many “hard programming problems” become ordinary engineering decisions.

End of the C & C++ Engineering Edition