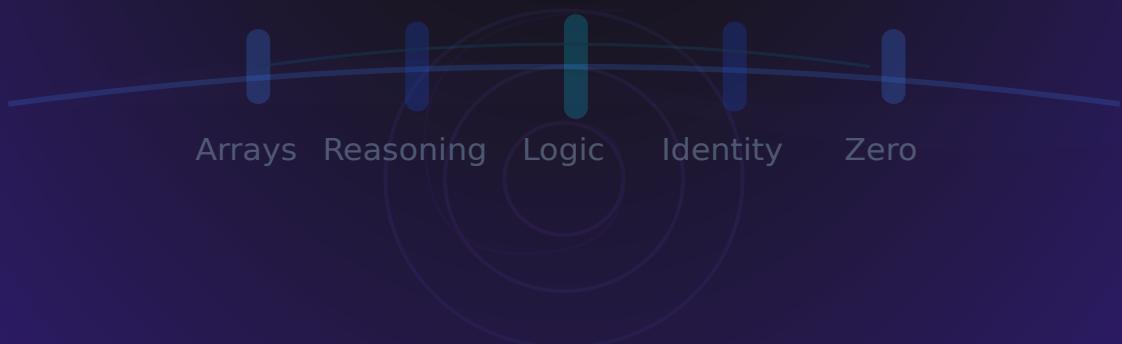


ARLIZ

A JOURNEY THROUGH ARRAYS



LIVING FIRST EDITION



A R L I

ARRAYS • REASONING • LOGIC • IDENTITY • ZERO

*"From ancient counting stones to quantum algorithms—
every data structure tells the story of human ingenuity."*

LIVING FIRST EDITION

Updated December 5, 2025

© 2025 Mahdi

CREATIVE COMMONS • OPEN SOURCE

LICENSE & DISTRIBUTION

ARLIZ: ARRAYS, REASONING, LOGIC, IDENTITY, ZERO

A Living Architecture of Computing

ARLIZ is released under the **Creative Commons Attribution-ShareAlike 4.0 International License** (CC BY-SA 4.0), embodying the core principles that define this work:

— Core Licensing Principles —

Arrays: *Structured sharing* — This work is organized for systematic access and distribution, like elements in an array.

Reasoning: *Logical attribution* — All derivatives must maintain clear reasoning chains back to the original work and author.

Logic: *Consistent application* — The same license terms apply uniformly to all uses and modifications.

Identity: *Preserved authorship* — The identity and contribution of the original author (Mahdi) must be maintained.

Zero: *No restrictions beyond license* — Starting from zero barriers, with only the essential requirements for attribution and share-alike.

FORMAL LICENSE TERMS

Copyright © 2025 Mahdi

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

License URL: <https://creativecommons.org/licenses/by-sa/4.0/>

You are free to:

- **Share** — copy and redistribute the material in any medium or format for any purpose, even commercially.
- **Adapt** — remix, transform, and build upon the material for any purpose, even commercially.

Under the following terms:

- **Attribution** — You must give appropriate credit to Mahdi, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
- **No additional restrictions** — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

DISTRIBUTION & SOURCE ACCESS

Repository: The complete source code (LaTeX, diagrams, examples) is available at:

<https://github.com/m-mdy-m/Arliz>

Preferred Citation Format:

Mahdi. (2025). *Arliz*. Retrieved from <https://github.com/m-mdy-m/Arliz>

Version Control: This is a living document. Check the repository for the most current version and revision history.

WARRANTIES & DISCLAIMERS

No Warranty: This work is provided "AS IS" without warranty of any kind, either expressed or implied, including but not limited to the implied warranties of merchantability and fitness for a particular purpose.

Limitation of Liability: In no event shall Mahdi be liable for any direct, indirect, incidental, special, exemplary, or consequential damages arising from the use of this work.

Educational Purpose: This work is intended for educational and research purposes. Practical implementation of algorithms and techniques should be thoroughly tested and validated for production use.

TECHNICAL SPECIFICATIONS

Typeset with: L^AT_EX using Charter and Palatino font families

Graphics: TikZ and custom illustrations

Standards: Follows academic publishing conventions

Encoding: UTF-8 with full Unicode support

Format: Available in PDF, and LaTeX source formats

License last updated: December 5, 2025

For questions about licensing, contact: bitsgenix@gmail.com

Contents

Title Page	i
Contents	iii
Preface	vi
Preface	vi
Acknowledgments	xii
How to Read This Book	xiii
Introduction	xxi
I Data Representation	1
1 The Philosophy of Representation	3
1.1 Why Representation Matters	3
1.2 The Beginning: Counting Before Numbers	4
1.2.1 How Ancient People Counted	4
1.2.2 From Concrete to Abstract	4
1.3 Ancient Philosophy Meets Representation	5
1.3.1 Plato's Cave and the Problem of Reality	5
1.3.2 Aristotle Takes a More Practical View	6
1.3.3 Medieval Philosophers Keep the Discussion Going	6
1.4 Modern Philosophy Sharpens the Questions	7
1.4.1 Descartes and the Mind	7
1.4.2 Kant's Radical Claim	7
1.4.3 Sartre on Imagination and Absence	8
1.5 What Is Representation, Really?	8
1.6 The Abstraction Hierarchy	10
1.6.1 What Abstraction Means	10
1.6.2 Layers in Computing Systems	10
1.6.3 Why This Matters for Arrays	11
1.7 Information Theory: The Mathematics of Representation	11
1.7.1 Shannon's Breakthrough	11
1.7.2 Information and Representation	12

1.7.3	Encoding and Decoding	12
1.7.4	Information Content and Surprise	13
2	Voltage to Bits	14
3	Binary Representation and Boolean Algebra	15
4	Number Systems and Bases	16
5	Integer Representation: Unsigned	17
6	Signed Integer Representation	18
7	Integer Overflow and Wraparound Behavior	19
8	Fixed-Point Representation	20
9	Floating-Point Representation: IEEE 754	21
10	Special Floating-Point Values	22
11	Floating-Point Arithmetic Operations	23
12	Floating-Point Error Analysis	24
13	Decimal Floating-Point Representation	25
14	Extended Precision and Arbitrary Precision	26
15	Character Encoding: ASCII and Extensions	27
16	Unicode: Universal Character Encoding	28
17	Unicode Transformation Formats	29
18	Text Processing Complexities	30
19	Endianness: Byte Ordering	31
20	Cross-Platform Data Exchange	32
21	Bitwise Operations Fundamentals	33
22	Bit Shifting and Rotation	34
23	Bit Manipulation Techniques	35
24	Bit Packing and Flags	36
25	Advanced Bit Hacking	37
26	Data Alignment Fundamentals	38
27	Structure Padding and Layout	39

28 Alignment Optimization Techniques	40
29 Color Representation Models	41
30 Pixel Formats and Bit Depth	42
31 Image Compression Fundamentals	43
32 Video Encoding Principles	44
33 Audio Representation: Sampling Theory	45
34 Audio Encoding Formats	46
35 Signal Processing Representations	47
36 Pointer Representation	48
37 Pointer Arithmetic and Address Calculation	49
38 Special Pointer Values	50
39 Error Detection Codes	51
40 Error Correction Codes	52
41 Cyclic Redundancy Checks (CRC)	53
42 Data Serialization Fundamentals	54
43 Binary Serialization Formats	55
44 Text Serialization Formats	56
45 Custom Binary Protocols	57
46 Data Compression Theory	58
47 Dictionary-Based Compression	59
48 Entropy Coding	60
49 Modern Compression Algorithms	61
50 Specialized Compression	62
Glossary	63
Bibliography & Further Reading	63
Reflections at the End	64
Index	66

Preface

EVERY BOOK HAS ITS ORIGIN STORY, and this one is no exception. If I were to capture the essence of creating this book in a single word, that word would be **curiosity**—though *improvised* comes as a close second. What you hold in your hands (or view on your screen) is the result of years of persistent questioning, a journey that began with a simple yet profound realization: I didn't truly understand what an array was.

This might sound trivial to some. After all, arrays are fundamental to programming, covered in every computer science curriculum, explained in countless tutorials. Yet despite encountering terms like `array`, `stack`, `queue`, `linked list`, `hash table`, and `heap` repeatedly throughout my studies, I found myself increasingly frustrated by the superficial explanations typically offered. Most resources assumed you already knew what these structures fundamentally represented—their conceptual essence, their implementation mechanics, their performance characteristics.

But I wanted the *roots*. I needed to understand not just how to use an array, but what it truly meant at every level—from hardware representation to high-level abstractions. This led me to a decisive moment:

If I truly want to understand, I must build from the foundation.

And so began the journey that became Arliz.

The Name and Its Meaning

The name "Arliz" started as a somewhat arbitrary choice—I needed a title, and it sounded right. However, as the book evolved, I discovered a fitting expansion that captures its essence:

Arliz = Arrays, Reasoning, Logic, Identity, Zero

This backronym embodies the core pillars of our exploration:

- **Arrays:** The fundamental data structure we seek to understand from implementation to application

- **Reasoning:** The systematic thinking behind data organization and algorithmic design
- **Logic:** The formal principles that govern computation and data manipulation
- **Identity:** The concept of distinguishing, indexing, and assigning meaning to elements within structures
- **Zero:** The foundation from which all indexing, computation, and systematic organization originates

You may pronounce it "Ar-liz," "Array-Liz," or however feels natural to you. I personally say "ar-liz," but the pronunciation matters less than the journey it represents.

The Genesis of This Work

This book was not conceived in its current form. Originally, Arliz was intended to be a comprehensive seven-part exploration spanning:

1. Philosophical and Historical Foundations
2. Mathematical Fundamentals
3. Data Representation
4. Computer Architecture and Logic
5. Array Odyssey
6. Data Structures and Algorithms
7. Parallelism and Systems

As I delved deeper into writing the first two parts—covering the historical evolution of counting systems and the mathematical prerequisites for understanding data structures—I confronted an uncomfortable reality. These sections were becoming substantial works in their own right, yet they were foundational not only for arrays but for understanding algorithmic analysis and computational thinking more broadly.

Simultaneously, I realized that array analysis alone could not stand without a proper treatment of algorithmic complexity. Understanding why an array operation is $O(1)$ or $O(n)$ requires deep analytical foundations that extend far beyond arrays themselves.

This led to a critical decision: rather than compromise the depth of treatment by constraining everything within a single volume, I would separate the foundational material into dedicated works. Thus emerged:

- **Mathesis: The Mathematical Foundations of Computing** — A comprehensive treatment of the mathematical concepts underlying all of computer science, from ancient number systems through modern discrete mathematics and linear algebra
- **The Art of Algorithmic Analysis** — A rigorous exploration of analytical techniques for understanding computational complexity, from asymptotic notation through advanced amortized analysis and complexity theory

These books were not afterthoughts or supplements—they became the essential prerequisites that enable Arliz to focus purely on what it does best: a deep, implementation-focused exploration of arrays and their role in computing systems.

The current Arliz, therefore, begins where those foundations end. It assumes mathematical maturity at an intermediate level—comfort with discrete mathematics, basic linear algebra, and algorithmic analysis—and builds from there into the concrete realities of array implementation, optimization, and application.

What This Book Represents

Arliz is not a gentle introduction to programming, nor is it a purely theoretical treatment of data structures. Instead, it represents something more focused and, I believe, more valuable: a comprehensive technical exploration of the most fundamental data structure in computing, examined from every relevant angle.

This living work evolves continuously as I discover better explanations, uncover new implementation details, or recognize deeper connections between concepts. As long as I continue learning, Arliz will continue growing. Your engagement—through corrections, suggestions, and questions—makes you part of this evolution.

The structure reflects a deliberate progression through increasingly sophisticated understanding:

- **Data Representation** — How information is encoded in digital systems, from number systems to character encoding
- **Computer Architecture and Logic** — The hardware foundations that determine how arrays actually work
- **Array Odyssey** — Deep exploration of array implementation, behavior, and optimization
- **Data Structures and Algorithms** — How arrays enable other structures and algorithmic techniques
- **Parallelism and Systems** — Arrays in multi-threaded, distributed, and high-performance contexts

Prerequisites and Expectations

This book assumes you have completed (or are comfortable with) the material in:

- **Mathesis** — Mathematical foundations including discrete mathematics, linear algebra, and basic analysis
- **The Art of Algorithmic Analysis** — Asymptotic analysis, recurrence relations, and algorithmic complexity

Without these foundations, much of this book will be challenging. With them, it becomes a focused, deep dive into one of computing’s most elegant and essential abstractions.

We will not shy away from technical complexity. Array implementation touches hardware architecture, memory hierarchies, compiler optimizations, and operating system interfaces. Understanding arrays properly means understanding these layers and their interactions.

That said, I have worked to avoid unnecessary mathematical abstraction. While mathematical rigor appears where needed—particularly when analyzing performance characteristics or proving correctness properties—the focus remains practical: how do arrays actually work, why do they behave as they do, and how can we use them effectively?

My Approach and Principles

Throughout the writing process, I have maintained three core principles:

1. **Implementation Focus:** Every abstract concept is grounded in concrete implementation. You will see how arrays are actually represented in memory, how compilers optimize array operations, and how hardware characteristics influence performance.
2. **Visual Understanding:** Complex concepts are accompanied by diagrams, memory layouts, and visual representations. Arrays are inherently spatial structures—understanding them requires seeing their organization.
3. **Practical Code:** Nearly every topic includes working implementations that can be studied, modified, and adapted. Theory without implementation is incomplete; implementation without theory is fragile.

An important disclosure: many of the implementations in this book are my own constructions, built from first principles to demonstrate concepts clearly. Some may

run slower than heavily optimized production libraries—others may reveal surprising efficiencies. The goal is understanding, not necessarily optimal performance in every case.

About the Author

I am **Mahdi**, though you may know me by my online alias: *Genix*. At the time of writing, I am a Computer Engineering student, but more fundamentally, I am someone driven by a relentless need to understand the systems I work with at their deepest levels.

My relationship with computers has been one of continuous investigation—never satisfied with surface-level explanations, always pushing toward the foundational principles that make everything work. This book represents that drive crystallized into a focused exploration of arrays.

How to Use This Book

Arliz is freely available and open source. You can access the complete PDF, LaTeX source code, and related materials at:

<https://github.com/m-mdy-m/Arliz>

Each chapter includes carefully designed exercises and implementation challenges. These are not optional—they are essential components of the learning process. True understanding of arrays comes only through implementing them yourself, seeing how they break under stress, and discovering their performance characteristics through measurement.

I encourage you to approach this book as a collaborative effort. If you discover errors, have implementation insights, or develop optimizations worth sharing, please contribute. This book improves through community engagement.

A Living Technical Document

Finally, I want to be transparent about what you are engaging with. This is not a finished, polished textbook in the traditional sense. It is an evolving technical exploration, growing and improving as understanding deepens and new implementation techniques emerge.

You may encounter sections that could be clearer, implementations that could be more efficient, or explanations that could be more rigorous. This is intentional—Arliz

represents learning in progress, understanding in development. It invites you to participate in this process of refinement rather than simply consume its content.

I hope this book serves you well—whether you are building your first serious data structures, optimizing performance-critical systems, or simply satisfying intellectual curiosity about how arrays actually work. And if you learn something valuable, discover an error, or develop an insight worth sharing, I hope you will contribute.

After all, this book grows with all of us.

Mahdi
2025

"The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise."

— EDSGER W. DIJKSTRA

Acknowledgments

I would like to express my gratitude to everyone who supported me during the creation of this book. Special thanks to the open-source community for their invaluable resources and to all those who reviewed early drafts and provided feedback.

How to Read This Book

Understanding the Structure

Arliz is organized as a progressive deepening of understanding. Each part builds on previous material, developing increasingly sophisticated perspectives on arrays and their implementation. You cannot skip ahead without missing essential foundations.

The Five Parts

Part I: Data Representation

Begin here. Always. This part establishes how information is encoded in digital systems—the absolute foundation for understanding how array elements are stored and manipulated. Without this foundation, later material becomes incomprehensible.

Part II: Computer Architecture & Logic

Arrays live in hardware. This part examines that hardware: logic gates, memory systems, processor architecture, cache behavior. Understanding these realities is essential for writing efficient array code.

Part III: Array Odyssey

The core of the book. Here we explore arrays themselves: their mathematical properties, memory layouts, performance characteristics, optimization techniques. This is where everything converges.

Part IV: Data Structures & Algorithms

Arrays enable other structures. This part examines how stacks, queues, heaps, hash tables, and other structures build on array foundations. We also explore algorithmic techniques that leverage array properties.

Part V: Parallelism & Systems

Modern computing is parallel and distributed. This part examines arrays in multi-threaded, concurrent, and distributed contexts—showing how classical concepts extend to contemporary challenges.

Reading Strategies

Sequential Reading (Recommended)

For most readers, sequential reading provides the best experience. Start with Part I, work through each chapter carefully, complete the exercises, implement the examples, and progress to the next part only when you have mastered the current material.

This approach takes time—months, not weeks—but produces deep, lasting understanding. Concepts build systematically. Each part prepares you for the next. Skipping ahead creates gaps that will eventually force you to backtrack.

Selective Reading (For Experienced Readers)

If you have strong backgrounds in both computer architecture and mathematical analysis, you might consider selective reading:

- **Part I:** Review chapter summaries. If material seems familiar, proceed to Part II. If anything seems unclear, read the full chapters.
- **Part II:** Same approach—review summaries, read full chapters for unfamiliar material.
- **Part III:** Read completely. This is the core material that justifies the book's existence.
- **Parts IV-V:** Read sequentially for complete understanding, or select chapters based on your specific interests.

Be honest with yourself about your preparation. Overestimating your background leads to gaps that undermine later understanding.

Reference Use

Once you have read the book completely, it serves as a reference. The detailed table of contents, comprehensive index, and clear section organization enable targeted consultation when specific questions arise.

But initial reading should be sequential. Reference use only becomes effective after establishing comprehensive understanding.

Engaging with the Material

Work Every Example

Examples are not illustrations—they are essential learning tools. For each example:

1. Read the example carefully, ensuring you understand each step
2. Implement the example in your preferred programming language
3. Run the implementation and verify it produces expected results
4. Modify the example to test your understanding
5. Measure performance characteristics when relevant

Understanding comes through implementation, not merely reading.

Complete the Exercises

Exercises test and deepen understanding. They range from straightforward verification of concepts through challenging implementation problems to open-ended research questions.

- **Basic exercises:** Verify you understand fundamental concepts
- **Intermediate exercises:** Apply concepts to new situations
- **Advanced exercises:** Extend concepts in novel directions
- **Research problems (★):** Open questions requiring substantial investigation

Do not skip exercises. They are not optional—they are core components of the learning process.

Measure Performance

Array performance is empirical. Throughout the book, we make performance predictions based on theoretical analysis. Verify these predictions through measurement:

1. Implement the operation being analyzed
2. Measure its actual performance using timing tools
3. Compare measurements to theoretical predictions
4. Investigate and explain any discrepancies
5. Vary parameters to observe how behavior changes

This empirical engagement develops performance intuition that no amount of reading can provide.

Question Everything

Active questioning drives deeper understanding. As you read:

- Why does this operation have this cost?
- How would changing this parameter affect behavior?
- What hardware characteristics influence this performance?
- Could this technique be implemented differently?
- What are the trade-offs in this design decision?
- How does this concept connect to material in other chapters?

When you find yourself unable to answer such questions, that indicates areas requiring deeper study.

Prerequisites and Preparation

Essential Mathematical Background

You should be comfortable with:

- **Discrete Mathematics:** Sets, relations, functions, graph theory, combinatorics
- **Linear Algebra:** Vectors, matrices, linear transformations
- **Mathematical Analysis:** Asymptotic notation, series, limits
- **Probability:** Basic probability theory, random variables, expected values

If these topics seem unfamiliar, work through *Mathesis: The Mathematical Foundations of Computing* before continuing with *Arliz*.

Essential Algorithmic Background

You should understand:

- **Asymptotic Analysis:** Big-O, Big-Omega, Big-Theta notation
- **Recurrence Relations:** Solving recurrences, Master theorem
- **Algorithm Analysis:** Analyzing time and space complexity
- **Basic Data Structures:** Conceptual understanding of lists, trees, graphs

If this material seems unfamiliar, work through *The Art of Algorithmic Analysis* before continuing with *Arliz*.

Programming Proficiency

You should:

- Be proficient in at least one programming language (C, C++, Java, or Python recommended)
- Be comfortable reading code in multiple languages
- Understand basic computer architecture concepts (CPU, memory, registers)
- Have experience implementing and debugging non-trivial programs

This book is not for beginners. It assumes substantial programming experience.

Notation and Conventions

Mathematical Notation

We use standard mathematical notation throughout:

- $\mathcal{O}(f(n))$: Big-O notation for asymptotic upper bounds
- $\Omega(f(n))$: Big-Omega notation for asymptotic lower bounds
- $\Theta(f(n))$: Big-Theta notation for tight asymptotic bounds
- $[n]$: The set $\{1, 2, \dots, n\}$
- $\log n$: Logarithm base 2 unless otherwise specified

Pseudocode Conventions

Pseudocode uses clear, imperative style:

- Array indexing starts at 0 unless explicitly stated otherwise
- $A[i]$ accesses element at index i of array A
- \leftarrow denotes assignment
- Loops use clear indentation to show scope
- Comments appear in italic type

Implementation Examples

Code examples appear in monospace font with syntax highlighting:

- Primary examples use C for clarity and control

- Alternative implementations may appear in C++, Java, or Python
- Assembly code appears when discussing low-level implementation
- All code is complete and executable unless marked as pseudocode

Common Pitfalls to Avoid

Skipping Mathematical Development

Mathematical analysis is not optional decoration—it is essential substance. When a theorem appears, read its proof carefully. When an analysis uses mathematical techniques, work through the mathematics. This rigor distinguishes genuine understanding from surface familiarity.

Reading Without Implementing

Reading about arrays is not the same as implementing array operations. Reading about cache behavior is not the same as measuring cache performance. Reading about optimization techniques is not the same as applying those techniques to real code.

Implementation is not optional. Do the work.

Ignoring Hardware Realities

Arrays do not exist in abstract mathematical space—they exist in physical hardware with specific characteristics and constraints. Cache lines, memory alignment, TLB behavior, SIMD instructions—these realities determine actual performance. Understanding them is essential.

Settling for Vague Understanding

When you find yourself thinking "I sort of understand this," stop. That signals insufficient understanding. Go back. Read again. Implement examples. Work exercises. Achieve precise, confident understanding before proceeding.

Vague understanding compounds over chapters, eventually producing complete confusion.

When You Get Stuck

Getting stuck is normal—it signals you have encountered material requiring deeper engagement. When stuck:

1. Return to the previous chapter and verify you truly understood that material
2. Re-read the challenging section carefully, taking notes
3. Implement examples from the section
4. Work through exercises, even if they seem difficult
5. Consult the references cited in the chapter
6. Take a break and return with fresh perspective

Persistent difficulty despite these strategies may indicate insufficient prerequisites. Be honest with yourself—return to *Mathesis* or *The Art of Algorithmic Analysis* if necessary.

Using This Book as a Course Text

Undergraduate Course

For an undergraduate course on data structures or advanced programming:

- Parts I-III provide core material for a semester-long course
- Part IV material can be integrated as time permits
- Part V provides advanced material for motivated students
- Exercises provide abundant homework and project material

Graduate Course

For a graduate course on advanced data structures or performance optimization:

- Assume students have mastered prerequisites
- Part III provides core material
- Parts IV-V provide substantial advanced material
- Research problems (*) provide thesis-level investigations

Final Advice

This book rewards patience, persistence, and active engagement. It punishes skimming, skipping, and passive reading.

Take your time. Work through examples. Implement techniques. Measure performance. Question constantly. When concepts seem difficult, that is normal—persist until understanding comes.

The journey requires sustained effort, but the destination—deep, rigorous understanding of the most fundamental data structure in computing—justifies every moment invested.

Welcome to *Arliz*. Begin when ready.

Introduction

ARRAYS ARE EVERYWHERE. Every image you view, every text you read, every game you play, every database query you execute—arrays underlie them all. Yet despite their ubiquity, arrays remain poorly understood by most programmers. They are treated as primitive constructs, learned hastily and used mechanically, their true nature obscured by layers of abstraction.

This book exists to remedy that situation.

What This Book Is

Arliz is a focused, technical exploration of the most fundamental data structure in computing: the array. Unlike comprehensive data structures textbooks that survey many structures superficially, we examine arrays with unprecedented depth—from their hardware representation in memory through their implementation in programming languages to their role in advanced algorithmic techniques.

This is not a book for casual reading. It demands engagement, rewards persistence, and assumes substantial mathematical and computational maturity. If you seek quick tutorials or surface-level explanations, you will find this book frustrating. If you seek deep, rigorous understanding of how arrays actually work and why they behave as they do, you have found the right resource.

The Architecture of This Work

The book progresses through five major parts, each building on the foundations established by its predecessors:

Part I: Data Representation

We begin with the fundamental question: how is information encoded in digital systems? From binary representation through character encoding, from integer formats to floating-point arithmetic, we establish the representational foundations that determine how array elements are actually stored and manipulated at the machine level.

Part II: Computer Architecture & Logic

Arrays do not exist in abstract space—they live in physical hardware with specific characteristics and constraints. We examine logic gates, processor architecture, memory hierarchies, and cache behavior. Understanding these hardware realities is essential for writing array code that performs well on real systems.

Part III: Array Odyssey

This is the heart of the book. We explore arrays from every angle: their mathematical properties, their implementation in memory, their performance characteristics, their optimization techniques. We examine one-dimensional arrays, multidimensional arrays, sparse arrays, and specialized array variants. We analyze cache behavior, memory alignment, and hardware-level optimization strategies.

Part IV: Data Structures & Algorithms

With arrays deeply understood, we examine how they enable other data structures. Stacks, queues, heaps, hash tables—all build on array foundations. We explore algorithmic techniques that leverage array properties, from sorting algorithms through dynamic programming.

Part V: Parallelism & Systems

Modern computing is parallel and distributed. We examine how arrays behave in multi-threaded environments, how they scale across distributed systems, and how they enable high-performance computing. This part connects classical array concepts to cutting-edge computational challenges.

What Makes This Different

Several characteristics distinguish this treatment:

Implementation Focus We do not merely describe abstract properties—we examine actual implementations. You will see precisely how arrays are represented in memory, how compilers optimize array operations, and how hardware characteristics influence performance.

Performance Analysis Every operation receives rigorous performance analysis. We measure cache behavior, count memory accesses, and analyze asymptotic complexity. Understanding why operations cost what they cost is central to using arrays effectively.

Mathematical Rigor Arrays are mathematical objects with well-defined properties. We develop this mathematics carefully, proving theorems about array behavior and analyzing operations with formal precision.

Hardware Awareness Arrays cannot be understood independently of the hardware that implements them. We examine how memory systems work, how caches behave, and how processors optimize array access patterns.

Prerequisites

This book assumes substantial preparation:

Mathematical Maturity: You should be comfortable with discrete mathematics, linear algebra, and basic mathematical analysis. Specifically, you should have completed (or be comfortable with) the material in *Mathesis: The Mathematical Foundations of Computing*.

Algorithmic Analysis: You should understand asymptotic notation, recurrence relations, and basic complexity analysis. The material in *The Art of Algorithmic Analysis* provides the necessary foundations.

Programming Experience: You should be proficient in at least one programming language and comfortable reading code in multiple languages. We use pseudocode, C, and occasionally other languages for examples.

Without these prerequisites, you will struggle with substantial portions of this book. With them, you are prepared for a deep, rewarding exploration of arrays.

How to Approach This Material

This book rewards active engagement:

Work Through Examples: Every example can be implemented and experimented with. Do so. Understanding comes through doing, not merely reading.

Measure Everything: Array performance is empirical. Write code, measure its behavior, and compare measurements to theoretical predictions. This develops intuition no amount of reading can provide.

Question Constantly: Why does this operation cost what it costs? How would changing this parameter affect performance? What hardware characteristics influence this behavior? Active questioning drives deeper understanding.

Implement Techniques: The optimization techniques we discuss should be implemented and tested. Theory becomes meaningful only when connected to practice.

The Living Nature of This Work

Like all my books, *Arliz* evolves continuously. As I discover better explanations, identify errors, recognize new connections, or encounter new implementation techniques, the book improves. Your engagement—through corrections, suggestions, implementations, and questions—contributes to this evolution.

Check the GitHub repository regularly for updates. The version you read today will be superseded by improved versions tomorrow. This is not a weakness but a strength—the book remains current, accurate, and relevant.

A Note on Difficulty

This book is challenging by design. Arrays may seem simple on the surface, but their full understanding requires grappling with hardware architecture, memory systems, compiler optimizations, and algorithmic analysis. Some sections will require sustained effort to master.

This difficulty is intentional. Genuine understanding is never cheap—it demands intellectual investment, persistent effort, and willingness to struggle with complex concepts. If you find sections difficult, that is normal. Persist. Work through examples. Implement the techniques. Understanding will come through sustained engagement.

What You Will Gain

By the end of this journey, you will possess:

- **Deep Implementation Understanding:** Precise knowledge of how arrays are represented, stored, and manipulated at every level from hardware through high-level languages
- **Performance Intuition:** Ability to predict array performance characteristics and optimize array-based code effectively
- **Algorithmic Capability:** Understanding of how arrays enable algorithmic techniques and data structure implementations
- **Hardware Awareness:** Knowledge of how memory hierarchies, caches, and processors influence array behavior
- **Design Insight:** Ability to choose appropriate array representations and implementations for specific problems

More fundamentally, you will have developed a way of thinking about data structures that transcends arrays themselves. The analytical techniques, performance

reasoning, and implementation understanding you develop here transfer to all subsequent work with computational systems.

Begin

Five parts await. Each deepens your understanding of arrays—from representation through implementation to application. Each builds essential capabilities for working effectively with the most fundamental data structure in computing.

Welcome to **Arliz**. Let us explore arrays with the depth and rigor they deserve.

“Simplicity is prerequisite for reliability.”

— EDSGER W. DIJKSTRA

“The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise.”

— EDSGER W. DIJKSTRA

“Premature optimization is the root of all evil.”

— DONALD E. KNUTH

Part I

Data Representation

BEFORE WE can understand how arrays store elements, we must first understand how computers represent information itself. Every piece of data—numbers, text, images, instructions—exists as patterns of bits. This part explores the foundational question: how do we encode meaning into binary?

“The choice of representation is often more important than the choice of algorithm.”

— DONALD E. KNUTH

Chapter 1

The Philosophy of Representation

Before we dive into arrays and how computers actually store stuff, we need to ask something more basic: what does it even mean to *represent* something? This chapter is about that question. I know it might seem weird to start a book about arrays with philosophy, but trust me—we need this foundation. If we don't understand representation itself, we can't really get how arrays work or why they matter.

1.1 Why Representation Matters

Let me start simple. When you see the number "5" written on paper, what are you actually looking at? You're looking at ink, right? Some dark marks on a light surface. But somehow, those marks *mean* something. They represent the idea of five-ness—the concept of having five of something.

This is weird when you actually think about it.

Computers don't work with real numbers or letters or pictures. They work with electrical signals. High voltage, low voltage. That's literally it. But somehow we use those electrical signals to represent *anything*—numbers, words, pictures, music, this entire book you're reading right now.

This is representation at work.

Here's the key idea: **representation is when one thing stands for another thing**. The symbol "5" stands for the number five. A high voltage in computer memory stands for the binary digit 1. A specific pattern of bits stands for the letter "A." Everything in computing is representation, all the way down.

And arrays? Arrays are just organized representations. They're a way of representing many pieces of data in order, one after another, in memory. But before we get there,

we need to understand where this whole idea of representation came from and why it matters.

1.2 The Beginning: Counting Before Numbers

1.2.1 How Ancient People Counted

Humans have been representing things for a very long time—way longer than you might think. Long before writing existed, long before number systems as we know them, people needed to keep track of stuff. How many sheep do I have? How many days until the next full moon? How much grain is stored for winter?

The earliest evidence we have shows people using physical objects to represent quantities. They used stones, sticks, marks carved into bones [ifrah2000universal](#). Each stone represents one sheep. Each notch represents one day. This is called **tallying**, and it's the most basic form of representation.

One of the oldest examples is the Lebombo bone, found in the Lebombo Mountains between South Africa and Eswatini. It's a baboon fibula with 29 deliberate notches carved into it, dated to around 35,000 BCE [marshack1991roots](#). We don't know exactly what those notches represented—maybe lunar phases, maybe days, maybe something else—but the point is clear: one mark equals one thing. This is concrete, one-to-one representation.

Another famous example is the Ishango bone from the Democratic Republic of Congo, dated to around 20,000 BCE. It has notches arranged in three columns with distinct patterns [pletser2012ishango](#). Some researchers think it shows early mathematical thinking—maybe even prime numbers. Whether that's true or not, what matters is that humans were using physical marks to represent abstract quantities thousands of years before writing existed.

1.2.2 From Concrete to Abstract

Something really important happened over thousands of years: people moved from **concrete representation to abstract representation** [schmandt1996prehistory](#).

What's the difference?

Concrete representation: This specific stone represents *this specific* sheep. That particular mark represents *that particular* day.

Abstract representation: The symbol "5" represents the *concept* of five-ness, regardless of whether we're talking about sheep, days, apples, or anything else.

This shift from concrete to abstract is huge. It's one of the most important developments in human cognitive history [dehaene2011number](#). And it didn't happen overnight—it took thousands of years and happened gradually as human societies

became more complex.

According to Denise Schmandt-Besserat's research on ancient Mesopotamian tokens, this transition can be traced archaeologically [schmandt1996prehistory](#). Around 8000 BCE, people in the Middle East started using clay tokens—small shaped objects—to represent commodities. A cone-shaped token represented a small measure of grain. A sphere represented a large measure. This was still fairly concrete: each token represented a specific quantity of a specific thing.

But around 3500 BCE, something changed. Instead of using actual tokens, people started making impressions of the tokens in clay tablets. The *symbol* of the cone replaced the actual cone token. This is abstraction—the mark on clay isn't the grain, and it isn't even the token that represented the grain. It's a representation of a representation.

Peter Damerow and Robert Englund argue that this abstraction happened because of practical economic needs [damerow1987prehistory](#). As societies grew, managing resources became more complex. Simple tallying wasn't enough anymore. You needed systems that could represent quantities, track transactions, and keep records over time. Abstract number systems emerged from this practical necessity.

1.3 Ancient Philosophy Meets Representation

1.3.1 Plato's Cave and the Problem of Reality

The ancient Greek philosopher Plato told a famous story that's actually about representation, even though that's not usually how it's taught. It's called the Allegory of the Cave, from his work *The Republic* [plato1991republic](#).

Here's the story: Imagine people chained in a cave since childhood, facing a wall. Behind them is a fire, and between them and the fire, other people walk by carrying objects. The chained people can only see shadows cast on the wall in front of them. To these prisoners, the shadows *are* reality. They don't know about the actual objects casting the shadows, about the fire providing light, or about the world outside the cave.

One day, a prisoner is freed and dragged outside into sunlight. At first, he's blinded and confused. But gradually, he sees real objects, not just shadows. He realizes that everything he thought was real was just a representation—a poor copy of actual reality.

Plato was making a point about knowledge and truth, but he was also deeply worried about representation [fine1999plato](#). He thought that most of what we experience in the physical world are just representations—imperfect copies—of perfect Forms that exist in some abstract realm. A triangle you draw on paper isn't a *real* triangle

according to Plato; it's just a flawed representation of the perfect Form of Triangle. For our purposes, what matters is this: **Plato understood that representations can be misleading.** They show you something without showing you everything. A shadow on the wall gives you information about an object, but it's incomplete information. The shadow is two-dimensional; the object is three-dimensional. The shadow lacks color, texture, substance.

This is important when we think about how computers represent information. When you store a number in computer memory, you're not storing the abstract mathematical concept of that number. You're storing a pattern of electrical charges that *represents* that number according to a specific encoding scheme. The representation has limitations—it can overflow, lose precision, or be misinterpreted if you use the wrong decoding scheme.

1.3.2 Aristotle Takes a More Practical View

Aristotle, who studied under Plato, had a different approach **shields2016aristotle**. He wasn't obsessed with perfect Forms in some abstract realm. Instead, he studied how things actually work in the real world, including how we think and know things. In his work *De Anima* (On the Soul), Aristotle discussed how we perceive objects **aristotle2016soul**. When you see a tree, your mind doesn't contain the actual tree—that would be impossible. Instead, your mind receives the *form* of the tree through your senses. The form is like a pattern or structure. Your mind creates an internal representation based on this form.

Aristotle's idea is more practical than Plato's: representations in our minds are tools for navigating reality. They don't have to be perfect copies; they just need to be useful enough to let us interact with the world.

This practical view is actually closer to how we think about representations in computing. We don't need perfect representations—we need representations that are good enough for our purposes and that we can actually work with.

1.3.3 Medieval Philosophers Keep the Discussion Going

Medieval thinkers, especially Thomas Aquinas in the 13th century, continued developing these ideas about representation **pasnau2002thomas**.

Aquinas, working within the Christian philosophical tradition, tried to reconcile Aristotle's ideas with Christian theology. In his massive work *Summa Theologica*, he discussed how human knowledge works **aquinas1948summa**. According to Aquinas, when you perceive something—say, a tree—the form or "species" of that tree is received by your senses and then processed by your mind. You end up with a mental representation that allows you to think about trees even when no tree is

actually present.

Why am I telling you about medieval philosophy in a book about arrays? Because these thinkers were wrestling with exactly the problem we face in computing: **how can one thing stand for another thing?** How can patterns (whether in neurons or in computer circuits) represent something that isn't physically present?

The medieval philosophers understood something crucial: representation requires both a pattern (the representation itself) and an interpreter (something that can understand what the pattern means). This insight is fundamental to computer science. A sequence of bits doesn't mean anything on its own—it requires interpretation according to some encoding scheme.

1.4 Modern Philosophy Sharpens the Questions

1.4.1 Descartes and the Mind

In the 1600s, René Descartes basically restarted Western philosophy with his obsession over certainty **descartes1637discourse**. His famous statement "Cogito, ergo sum" (I think, therefore I am) came from his attempt to find something he could know with absolute certainty.

Descartes ended up focusing heavily on mental representations—ideas in the mind **descartes1641meditations**. He argued that when we think about anything, we have a mental representation or "idea" of it. These representations are distinct from the things they represent. You can think about a unicorn (you have a mental representation of it) even though unicorns don't exist in the physical world.

This might seem obvious now, but it was a big deal. Descartes established clearly that **representations can exist independently from what they represent**. The representation and the represented are separate things.

This is exactly what happens in computers. We have patterns of bits representing data, regardless of whether that data corresponds to anything in the physical world. You can represent the number -5 even though you can't have negative five apples. You can represent imaginary numbers, infinity, or undefined values—all as bit patterns in memory.

1.4.2 Kant's Radical Claim

In the late 1700s, Immanuel Kant made a radical argument: we never experience reality directly **kant1781critique**. Everything we experience is shaped and structured by our minds. We only ever experience our own mental representations, not things-in-themselves.

According to Kant, our minds don't passively receive information from the world.

Instead, our minds actively organize and structure sensory data according to built-in categories. Space, time, causality—these aren't features of reality itself according to Kant. They're structures our minds impose on experience.

When you see a red apple, you're not seeing the apple as it really is "in itself." You're seeing your mind's representation—a construction built from sensory data organized according to your mental categories.

This is relevant to computing because computers also actively construct representations **burks1981computer**. When you take a digital photo, the camera doesn't capture "the image." It samples light at millions of points, converts those measurements to numbers, and encodes those numbers as bits. The resulting file is a constructed representation, not a passive copy.

Understanding this helps us appreciate both the power and limitations of digital representations. A JPEG image isn't the scene itself, and it isn't even a neutral capture of the scene—it's a specific encoding optimized for human perception, with deliberate information loss.

1.4.3 Sartre on Imagination and Absence

Jean-Paul Sartre, writing in the 20th century, focused on imagination and what it means to represent something that isn't present **sartre2004imaginary**.

In his book *The Imaginary: A Phenomenological Psychology of the Imagination*, Sartre argued that when we imagine something, we're creating a special kind of representation—one that presents its object as *not being there* **sartre2004imaginary**. When you imagine a purple elephant, you know you're imagining. Your mental representation includes this "unreality marker."

According to Sartre, imagining isn't just a weak form of perceiving. It's a fundamentally different mental act. In perception, something presents itself as really there. In imagination, something presents itself as absent—as not-really-present.

This might sound abstract, but it matters for computing. In programming, we constantly represent things that aren't there. A null pointer represents the *absence* of a reference. A variable might be uninitialized, representing the absence of a value. An empty array represents a collection with no elements—absence of data.

Understanding that representations can encode absence as much as presence helps us design better data structures and write clearer code.

1.5 What Is Representation, Really?

After all this philosophy, let's get clear about what representation actually means.

Representation is when one thing (the representation) stands for another thing

(the represented) according to some system or convention.

A few key points to understand:

The representation is not what it represents The word "dog" is not a dog. If you write "fire" on paper, the paper doesn't burn. The number "42" stored in computer memory isn't the number forty-two in some abstract mathematical sense—it's a specific pattern of electrical charges that we interpret as forty-two.

This seems obvious, but people forget it. Programmers sometimes talk as if variables *are* values, when really variables are just locations in memory that *represent* or *hold* values.

Representation requires a system Why does "5" mean five? Because we have a convention—a shared agreement among humans—that this particular symbol represents this particular quantity **ifrah2000universal**.

In computing, we have many such conventions. ASCII is a convention that maps numbers to characters. Two's complement is a convention for representing negative integers. IEEE 754 is a convention for representing floating-point numbers **ieee2008754**.

These conventions are arbitrary in the sense that we could have chosen different ones. But once established, they become standards that everyone follows to ensure communication and compatibility.

Representations can be ambiguous The same representation can mean different things in different contexts. The byte 01000001 might represent the decimal number 65, or it might represent the letter 'A' in ASCII, or it might be part of a larger multi-byte sequence representing something else entirely.

The bits themselves don't determine meaning—the interpretation does. This is why type systems exist in programming languages: to keep track of how we should interpret different pieces of data.

Representations can be layered This is crucial: representations can represent other representations **newell1976computer**.

A letter is represented by a number (its ASCII code). That number is represented by a pattern of bits. Those bits are represented by voltage levels in circuits. Those voltages are produced by arrangements of transistors and electrons.

It's representations all the way down, until you hit actual physical reality (electrons and electromagnetic fields).

Understanding these layers helps us work at the right level of abstraction. When

writing application code, you usually think about letters and strings. But when debugging a binary file format, you might need to think about bytes and bits. When optimizing memory usage, you might need to think about how data is physically laid out.

1.6 The Abstraction Hierarchy

Now we get to something really important for understanding computing: the idea of abstraction layers.

1.6.1 What Abstraction Means

Abstraction means hiding details. When you use abstraction, you work with a simplified representation that hides underlying complexity [liskov1974programming](#). Think about driving a car. You press the gas pedal, and the car accelerates. You don't think about fuel injection rates, combustion timing, crankshaft rotation, or gear ratios. All those details are abstracted away behind the simple interface of the pedal. The pedal is a representation of a complex system.

Good abstraction is essential to managing complexity. You can't think about everything at once—you'd be overwhelmed. Abstraction lets you focus on what matters at your current level while trusting that lower levels work correctly.

1.6.2 Layers in Computing Systems

Computing is built on layers of abstraction, with each layer representing and hiding the complexity of the layer below it [tanenbaum2006structured](#).

Physical layer: At the bottom, you have actual electrons moving through silicon. This is the only layer that's physically "real"—everything else is representation.

Electrical layer: We represent electron behavior as voltage levels. High voltage (typically 3-5 volts) represents 1. Low voltage (near 0 volts) represents 0.

Digital logic layer: We represent voltage patterns as bits. A bit is just a symbol that can be 0 or 1, but it represents an electrical state.

Number layer: We represent quantities using patterns of bits. For example, the 8-bit pattern 00000101 represents the number 5 in binary.

Character layer: We represent text using numbers. The number 65 represents the letter 'A' in ASCII encoding [ascii1963](#).

Data structure layer: We represent organized collections of data using structures like arrays, linked lists, trees, and hash tables. An array represents an ordered sequence of values stored contiguously in memory.

Algorithm layer: We represent procedures and problem-solving methods using algorithms. An algorithm represents a step-by-step process for computing something.

Application layer: We represent user tasks and needs using application programs. A word processor represents the activity of writing and editing documents. Each layer builds on the layer below. Each layer hides the details of lower layers. And crucially: **each layer is a form of representation.**

1.6.3 Why This Matters for Arrays

Arrays exist at multiple levels in this hierarchy:

- **Physically:** An array is electrical charges stored in memory transistors
- **Electrically:** It's a pattern of voltages across memory cells
- **Digitally:** It's a sequence of bits in memory addresses
- **As a data structure:** It's an ordered collection of elements that can be accessed by index

When we work with arrays in code, we're working at the data structure level. We think about indices and elements and operations like reading or writing. We don't think about voltages or transistors.

But understanding that arrays are representations—that they exist in this hierarchy of abstractions—helps us understand their properties and limitations. It helps us know when to think about high-level operations and when to dig into lower-level details like memory layout and cache behavior.

1.7 Information Theory: The Mathematics of Representation

1.7.1 Shannon's Breakthrough

In 1948, Claude Shannon published a paper that changed everything: "A Mathematical Theory of Communication" [shannon1948theory](#). This paper established information theory—the mathematical study of information, communication, and representation.

Shannon's key insight was this: **information is fundamentally about reducing uncertainty.**

Think about it this way. Before I tell you something, you're uncertain about what I'll say. There are multiple possibilities. When I actually say something specific, I reduce your uncertainty. The amount of information in my message is directly related to how much uncertainty it removes.

Here's a simple example. Suppose I'm going to tell you the result of a coin flip.

Before I tell you, there are two equally likely possibilities: heads or tails. When I say "heads," I've eliminated one possibility and removed your uncertainty. I've given you one **bit** of information.

A bit—binary digit—is the fundamental unit of information. It's the amount of information needed to choose between two equally likely possibilities [shannon1948theory](#).

1.7.2 Information and Representation

Shannon's theory connects directly to representation. To represent something, you need enough information to distinguish it from other possibilities.

If you want to represent one of two things (heads or tails), you need 1 bit.

If you want to represent one of four things, you need 2 bits (00, 01, 10, 11).

If you want to represent one of eight things, you need 3 bits.

The general formula: to represent N equally likely possibilities, you need $\log_2(N)$ bits, rounded up to the nearest integer [cover2006elements](#).

This is fundamental to understanding how computers represent data. Every piece of data is encoded using some number of bits. How many bits? Enough to distinguish it from other possible values it could be.

For example, to represent one of 256 possible byte values, you need 8 bits. To represent one character from a 128-character ASCII set, you need 7 bits (though we typically use 8, with one bit unused or used for other purposes).

1.7.3 Encoding and Decoding

Information theory clarifies the relationship between representation and interpretation [cover2006elements](#).

Encoding is the process of converting something into a representation. We encode the number 5 as the bit pattern 00000101 (in 8-bit unsigned binary).

Decoding is the process of interpreting a representation to recover what it represents. We decode the bit pattern 00000101 as the number 5.

Crucially, encoding and decoding require agreement on the representation scheme. If we use different schemes, communication fails. If I encode text using UTF-8 and you try to decode it as ASCII, you'll get garbage for characters outside the ASCII range. This is why standards are so important in computing. ASCII, Unicode, IEEE 754, TCP/IP—these are all agreed-upon representation schemes. Without such standards, we couldn't share data reliably across different systems.

1.7.4 Information Content and Surprise

Not all messages contain the same amount of information. Shannon showed that information content depends on probability **shannon1948theory**.

If I tell you something you already knew was almost certain, I've given you very little information. "The sun rose this morning." That's almost no information, because you were already nearly certain it would happen.

If I tell you something surprising—something you thought was unlikely—I've given you a lot of information. "A meteor just destroyed Paris." That's high information content because it's unexpected.

Shannon formalized this with the concept of **entropy**. Information entropy measures the average information content of messages from a source. Higher entropy means more uncertainty and thus more information when that uncertainty is resolved **cover2006elements**.

For our purposes, what matters is this: the information content of a representation depends on context. The same representation might carry different amounts of information depending on what was expected or what was already known.

In the next chapters, we'll move from philosophy to physics and engineering. We'll see how representation actually works in physical computers. How do you represent information using electricity? How do you build circuits that process representations? How do you organize representations in memory?

But I hope this chapter has given you a solid foundation. When we talk about bits and bytes, voltages and logic gates, transistors and memory cells—remember: we're always talking about representation. One thing standing for another. Patterns that mean something because we've agreed they mean something.

That's the fundamental insight that makes computing possible. And that's what we'll build on as we dive deeper into how computers actually work and how arrays fit into all of this.

Chapter 2

Voltage to Bits

How transistors encode binary states, voltage levels, noise margins, signal integrity, why binary won over ternary.

Chapter 3

Binary Representation and Boolean Algebra

Two-state logic, Boolean operations, truth tables, De Morgan's laws, bit as fundamental unit.

Chapter 4

Number Systems and Bases

Positional notation, decimal, binary, octal, hexadecimal. Base conversion algorithms.
Historical development. Why different bases matter for different purposes.

Chapter 5

Integer Representation: Unsigned

Binary positional notation, range limitations, conversion algorithms, hexadecimal convenience.

Chapter 6

Signed Integer Representation

Sign-magnitude, one's complement, two's complement (why it won), arithmetic operations, overflow detection.

Chapter 7

Integer Overflow and Wraparound Behavior

Undefined behavior, modular arithmetic, detecting overflow, saturating arithmetic, language-specific behaviors.

Chapter 8

Fixed-Point Representation

Q-format notation, scaling factors, precision-range tradeoffs, embedded systems applications, fixed-point arithmetic.

Chapter 9

Floating-Point Representation: IEEE 754

Sign, exponent, mantissa encoding, normalized and denormalized numbers, single vs. double precision.

Chapter 10

Special Floating-Point Values

Infinity (positive and negative), NaN (quiet and signaling), signed zero, subnormal numbers.

Chapter 11

Floating-Point Arithmetic Operations

Addition, multiplication, division, rounding modes (round to nearest, toward zero, toward \pm), fused multiply-add.

Chapter 12

Floating-Point Error Analysis

Precision loss, catastrophic cancellation, machine epsilon, relative error, Kahan summation algorithm.

Chapter 13

Decimal Floating-Point Representation

IEEE 754-2008 decimal formats, financial computing requirements, densely packed decimal.

Chapter 14

Extended Precision and Arbitrary Precision

80-bit extended precision, quadruple precision (128-bit), arbitrary precision libraries (GMP, MPFR).

Chapter 15

Character Encoding: ASCII and Extensions

7-bit ASCII, extended ASCII variants, code pages, limitations for internationalization.

Chapter 16

Unicode: Universal Character Encoding

Code points, planes, combining characters, grapheme clusters, normalization forms (NFC, NFD, NFKC, NFKD).

Chapter 17

Unicode Transformation Formats

UTF-8 (variable length, backward compatible), UTF-16 (surrogate pairs), UTF-32 (fixed length), BOM issues.

Chapter 18

Text Processing Complexities

Grapheme vs. code point counting, case folding, locale-dependent operations, text segmentation.

Chapter 19

Endianness: Byte Ordering

Big-endian vs. little-endian, historical reasons, network byte order, bi-endian systems, byte swapping.

Chapter 20

Cross-Platform Data Exchange

Serialization formats, portable binary formats, protocol design considerations.

Chapter 21

Bitwise Operations Fundamentals

AND, OR, XOR, NOT operations, truth tables, bit manipulation primitives.

Chapter 22

Bit Shifting and Rotation

Logical shift, arithmetic shift, rotate left/right, applications to multiplication/division by powers of 2.

Chapter 23

Bit Manipulation Techniques

Setting/clearing/toggling bits, bit masks, extracting bit fields, counting set bits (population count).

Chapter 24

Bit Packing and Flags

Packing multiple boolean flags, bitfields in structs, union tricks, bit arrays.

Chapter 25

Advanced Bit Hacking

Finding rightmost set bit, isolating bit patterns, bit reversal, Gray code conversion.

Chapter 26

Data Alignment Fundamentals

Natural alignment, alignment requirements by type, misalignment penalties.

Chapter 27

Structure Padding and Layout

Compiler padding rules, struct member ordering, packing pragmas, cache line awareness.

Chapter 28

Alignment Optimization Techniques

Manual padding, alignment attributes, reordering for cache efficiency.

Chapter 29

Color Representation Models

RGB, RGBA, HSV, HSL, CMYK, YUV, color spaces, gamma correction.

Chapter 30

Pixel Formats and Bit Depth

1-bit, 8-bit indexed, 16-bit (RGB565), 24-bit, 32-bit (with alpha), HDR formats.

Chapter 31

Image Compression Fundamentals

Lossless (PNG, GIF) vs. lossy (JPEG), compression ratios, quality tradeoffs.

Chapter 32

Video Encoding Principles

Frame types (I, P, B), motion compensation, codecs (H.264, H.265, VP9, AV1).

Chapter 33

Audio Representation: Sampling Theory

Nyquist theorem, sample rate, bit depth, quantization noise, aliasing.

Chapter 34

Audio Encoding Formats

PCM (uncompressed), FLAC (lossless), MP3, AAC, Opus (lossy), perceptual coding.

Chapter 35

Signal Processing Representations

Time domain, frequency domain, spectrograms, wavelet transforms.

Chapter 36

Pointer Representation

How pointers are stored in memory, pointer size (32-bit vs. 64-bit), pointer tagging.

Chapter 37

Pointer Arithmetic and Address Calculation

Array element addressing, struct member offsets, pointer subtraction.

Chapter 38

Special Pointer Values

Null pointers, wild pointers, dangling pointers, pointer alignment requirements.

Chapter 39

Error Detection Codes

Parity bits (even/odd), checksums, CRC algorithms, hash-based integrity.

Chapter 40

Error Correction Codes

Hamming codes, Reed-Solomon codes, LDPC codes, convolutional codes.

Chapter 41

Cyclic Redundancy Checks (CRC)

CRC polynomials, CRC-8, CRC-16, CRC-32 standards, efficient implementation using tables, applications in data integrity.

Chapter 42

Data Serialization Fundamentals

Converting in-memory structures to byte streams, portability issues.

Chapter 43

Binary Serialization Formats

Protocol Buffers, FlatBuffers, Cap'n Proto, MessagePack, CBOR.

Chapter 44

Text Serialization Formats

JSON, XML, YAML, TOML, human-readable vs. machine-efficient tradeoffs.

Chapter 45

Custom Binary Protocols

Designing efficient binary formats, alignment considerations, versioning.

Chapter 46

Data Compression Theory

Information theory basics, entropy, lossless vs. lossy bounds.

Chapter 47

Dictionary-Based Compression

LZ77, LZ78, LZW, Lempel-Ziv family, dictionary construction.

Chapter 48

Entropy Coding

Huffman coding, arithmetic coding, asymmetric numeral systems (ANS).

Chapter 49

Modern Compression Algorithms

zlib, gzip, bzip2, LZMA, Zstandard, Brotli, compression levels.

Chapter 50

Specialized Compression

Run-length encoding, delta encoding, columnar compression, domain-specific methods.

Glossary

Reflections at the End

As you turn the final pages of **Arliz**, I invite you to pause—just for a moment—and look back. Think about the path you’ve taken through these chapters. Let yourself ask:

“Wait... what just happened? What did I actually learn?”

I won’t pretend to answer that for you. The truth is—**only you can**. Maybe it was a lot. Maybe it wasn’t what you expected. But if you’re here, reading this, something must have kept you going. That means something.

This book didn’t start with a grand plan. It started with a simple itch: **What even is an array, really?** What began as a curiosity about a “data structure” became something much stranger and—hopefully—much richer. We wandered through history, philosophy, mathematics, logic gates, and machine internals. We stared at ancient stones and modern memory layouts and tried to see the invisible threads connecting them.

If that sounds like a weird journey, well—yeah. It was.

This is Not the End

Arliz isn’t a closed book. It’s a snapshot. A frame in motion. And maybe your understanding is the same. You’ll return to these ideas later, years from now, and see new angles. You’ll say, “Oh. That’s what it meant.” That’s good. That’s growth. Everything you’ve read here is connected to something bigger—algorithms, networks, languages, systems, even the people who built them. There’s no finish line. And that’s beautiful.

From Me to You

If this book gave you something—an idea, a shift in thinking, a pause to wonder—then it has done its job. If it made you feel like maybe programming isn’t just code and rules, but a window into something deeper—then that means everything to me.

Thank you for being here.
Thank you for not skipping the hard parts.
Thank you for choosing to think.

One More Thing

You're not alone in this.
The Arliz project lives on GitHub, and the conversations around it will (hopefully) continue. If you spot mistakes, have better explanations, or just want to say hi—come by. Teach me something. Teach someone else. That's the best way to say thanks.
Knowledge grows in community.
So share. Build. Break. Rebuild.
Ask better questions.
And always, always—stay curious.

Final Words

Arrays were just the excuse.
Thinking was the goal.
And if you've started to think more clearly, more deeply, or more historically about what you're doing when you write code—then this wasn't just a technical book.
It was a human one.
Welcome to the quiet, lifelong joy of understanding.

This completes the first living edition of Arliz.

Thank you for joining this journey from zero to arrays, from ancient counting to modern computation.

The exploration continues...

Author's Notes and Reflections

On Naming Conventions and Creative Processes

I should confess something about my naming process: I tend to pick names first and find meaningful justifications later. Very scientific, I know! The name "Arliz" started as a random choice that simply felt right phonetically. Only after committing to it did I discover the backronym that now defines its meaning. This probably says something about my creative process—intuition first, rationalization second.

This approach extends beyond naming. Many aspects of this book emerged organically from curiosity rather than systematic planning. What began as personal notes to understand arrays evolved into a comprehensive exploration of computational thinking itself.

On Perfectionism and Living Documents

You should know that many of the algorithms presented in this book are my own implementations, built from first principles rather than copied from optimized sources. This means you might encounter code that runs slower than industry standards—or occasionally faster, when serendipity strikes.

Some might view this as a weakness, but I consider it a feature. The goal isn't to provide the most optimized implementations but to demonstrate the thinking process that leads to understanding. When you can reconstruct a solution from fundamental principles, you've achieved something more valuable than memorizing an optimal algorithm.

On Academic Formality and Personal Voice

You might notice that this book alternates between formal academic language and more conversational tones. This is intentional. While I respect the precision that formal writing provides, I also believe that learning happens best in an atmosphere of intellectual friendship rather than academic intimidation.

When I suggest you could "use this book as a makeshift heating device" if you find the approach ridiculous, I'm not being flippant—I'm acknowledging that not every approach works for every learner. Intellectual honesty includes admitting when your methods might not suit your audience.

On Scope and Ambition

The scope of this book—from ancient counting to modern distributed systems—might seem overly ambitious. Some might argue that such breadth necessarily sacrifices depth. I disagree, but I understand the concern.

My experience suggests that understanding connections between disparate fields often provides insights that narrow specialization misses. When you see arrays as part of humanity's broader intellectual project, you understand them differently than when you see them as isolated programming constructs.

That said, if you find the historical sections tedious or irrelevant, you have my permission to skip ahead. The book is designed to be valuable even when read non-sequentially.

On Errors and Imperfection

I mentioned that you'll find errors in this book. This isn't false modesty—it's realistic acknowledgment. Complex explanations, mathematical derivations, and code implementations inevitably contain mistakes, especially in a work that grows and evolves over time.

Rather than viewing this as a flaw, I encourage you to see it as an opportunity for engagement. When you find an error, you're not just identifying a problem—you're participating in the process of building better understanding. The best learning often

happens when we encounter and resolve contradictions.

On Time Investment and Expectations

When I suggest this book requires months rather than weekends to master, I'm not trying to inflate its importance. Complex concepts genuinely require time to internalize. Mathematical intuition develops gradually, through repeated exposure and active practice.

If you're looking for quick solutions to immediate programming problems, this book will frustrate you. If you're interested in developing the kind of deep understanding that serves you throughout your career, the time investment will prove worthwhile.

On Community and Collaboration

This book exists because of community—the open-source community that provides tools and resources, the academic community that develops and refines concepts, and the programming community that applies these ideas in practice.

Your engagement with this material makes you part of that community. Whether you find errors, suggest improvements, or simply work through the exercises thoughtfully, you're contributing to the collective understanding that makes books like this possible.

Final Reflection

Writing this book has been an exercise in understanding my own learning process. I've discovered that I learn best by building connections between disparate ideas, by understanding historical context, and by implementing concepts from scratch rather than accepting them as given.

Your learning process might be entirely different. Use what serves you from this book, adapt what needs adaptation, and don't hesitate to supplement with other resources when my explanations fall short.

The goal isn't for you to learn exactly as I did, but for you to develop your own path to deep understanding.

These notes reflect thoughts and observations that didn't fit elsewhere but seemed worth preserving. They represent the informal side of a formal exploration—the human element in what might otherwise seem like purely technical content.