

MODERN FORTRAN PROGRAMMING

FROM FUNDAMENTALS TO
HIGH-PERFORMANCE SCIENTIFIC COMPUTING



MODERN FORTRAN
(FORTRAN 2018 AND LATER)



OBJECT-ORIENTED
DESIGN



PARALLEL PROGRAMMING
WITH COARRAYS



C INTEROPERABILITY



NUMERICAL CORRECTNESS
AND RELIABILITY



HIGH-PERFORMANCE
OPTIMIZATION



— ROBERT C. BRENNAN —

Modern Fortran Programming

From Fundamentals to High-Performance Scientific Computing

Steve Publications

This book is available at <https://leanpub.com/modernfortranprogramming>

This version was published on 2026-08-26



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2026 Steve Publications

Contents

From Fundamentals to High-Performance Scientific Computing	1
Introduction	2
What You Will Learn	2
How to Use This Book	3
A Note on Standards and Compilers	4
Why Learn Fortran Today?	5
Chapter 1: The Story of Fortran – History, Evolution, and Modern Relevance	7
The First Programming Language – FORTRAN I and the Birth of Compilers	7
Standardization and Survival – From FORTRAN 66 to Fortran 90 . . .	8
The Modern Renaissance – Object-Oriented Features, Parallelism, and Beyond	10
Why Fortran Today – Supercomputers, Legacy Codebases, and New Projects	12
How This Book Is Organized – A Roadmap for the Reader	13
Chapter 2: Your First Fortran Programs – Structure, Compilation, Execution	15
The Anatomy of a Fortran Program – Source Files and Program Units	15
Setting Up Your Environment – Compilers and Tools	16
Hello World and Beyond – Free-Form Source and Basic Structure . . .	18
Compilation and Linking – From Source to Executable	20
Common First Errors and How to Fix Them	22
A Slightly Larger Example – Organizing Code from the Start	23
Chapter 3: Data, Types, and Numbers – Variables, Kinds, Precision, Literals	26
Declaring Variables – Types, Names, and Style	26

CONTENTS

The Intrinsic Data Types – Integer, Real, Complex, Logical, Character	26
Kinds and Numerical Precision – Choosing the Right Representation .	26
Literals, Constants, and Parameters – Exact Values in Code	26
Operators and Expressions – Arithmetic, Relational, Logical	26
Summary	27
Chapter 4: Control Flow and Iteration	28
Conditional Execution – IF Constructs and SELECT CASE	28
Looping with DO – Count-Controlled Repetition	28
Advanced Loop Patterns – WHILE-like Loops, Exit, Cycle, Nested Loops	28
Structured Programming Discipline – Avoiding GOTO	28
Algorithmic Examples – Search, Sort, and Iterative Methods	28
Summary	28
Chapter 5: Arrays and Array Operations	30
Declaring Arrays – Shape, Bounds, and Initialization	30
Multidimensional Arrays – Matrices and Beyond	30
Array Syntax and Whole-Array Operations – Vectorized Code	30
Slices, Sections, and Masked Arrays – Selective Access	30
Memory Layout and Performance – Column-Major Order and Cache	
Behavior	30
Summary	30
Chapter 6: Input and Output	32
List-Directed I/O – Simple Reading and Writing	32
Formatted I/O – Edit Descriptors and Precision Control	32
Internal Files – String Formatting and Parsing	32
Unformatted and Stream I/O – Binary Data and Portability	32
Robust Input Handling – Error Checking and User Interaction	32
Summary	32
Chapter 7: Procedures and Interfaces	34
Functions and Subroutines – Designing Reusable Procedures	34
Argument Passing and Intent Attributes – Clarity and Safety	34
Optional and Keyword Arguments – Flexible Interfaces	34
Explicit Interfaces and Interface Blocks – Compile-Time Checking . .	34
Generic Interfaces and Operator Overloading – Polymorphic Names .	34
Summary	35
Chapter 8: Modules and Program Organization	36

CONTENTS

What Modules Are For – Encapsulation and Code Sharing	36
Module Structure – Public, Private, and USE Associations	36
Submodules – Separating Interface from Implementation	36
Scope Rules and Name Resolution – Avoiding Collisions	36
Organizing Large Projects – Module Architecture Patterns	36
Chapter 9: Derived Types and Object-Oriented Fortran	38
Defining Derived Types – Structuring Complex Data	38
Type-Bound Procedures – Methods in Fortran	38
Polymorphism and the CLASS Keyword – Dynamic Dispatch	38
Abstract Types and Type Extension – Inheritance Hierarchies	38
Design Patterns with Modern Types – When OOP Helps	38
Summary	39
Chapter 10: Memory Management – Allocatables, Pointers, and Dynamic Data	40
Allocatable Variables – Dynamic Sizing and Lifecycle	40
Allocation Patterns – Arrays, Derived Types, and Error Handling	40
Pointers and Pointer Semantics – When They Make Sense	40
Common Memory Pitfalls – Leaks, Dangling References, Double Free	40
Modern Alternatives – Preferring Allocatables Over Pointers	40
Chapter 11: Parallel Programming with Coarrays	42
The Coarray Model – Images, Teams, and Shared Variables	42
Declaring and Using Coarrays – Basic Syntax	42
Synchronization – Event-Based Coordination with SYNC	42
Communication Patterns – Scatter, Gather, Reductions	42
Practical Parallel Programs – Real Examples and Pitfalls	42
Chapter 12: Interoperability – Connecting Fortran to the World	43
Why Interoperate – Ecosystems and Libraries	43
The ISO_C_BINDING Module – Types, Constants, and Procedures	43
Calling C from Fortran – Practical Examples	43
Calling Fortran from C – Exporting Interfaces	43
Beyond C – Python, Julia, and Other Language Bridges	43
Summary	43
Chapter 13: Numerical Correctness and IEEE Floating-Point Facilities	45
How Floating-Point Works – Representation and Limitations	45
Precision Selection and Kind Parameters – Getting It Right	45

IEEE Arithmetic Facilities – Querying and Controlling Behavior	45
Numerical Stability – Algorithms That Work in Practice	45
Reproducibility – Making Results Repeatable	45
Summary	46
Chapter 14: Performance Optimization and High-Performance Com-	
puting	47
Understanding Compiler Optimizations – Levels and Flags	47
Writing Vectorizable Code – What Compilers Can Accelerate	47
Memory Access Patterns – Locality, Stride, and Cache Efficiency . . .	47
Profiling and Benchmarking – Measuring Real Performance	47
HPC Considerations – Scaling to Large Systems	47
Chapter 15: Build Systems, Testing, and Modern Toolchains	49
The Fortran Package Manager (fpm) – Modern Builds Made Simple . .	49
CMake for Fortran Projects – Cross-Platform Build Automation	49
Traditional Makefiles – When They Still Make Sense	49
Testing Strategies – Unit Tests and Validation Frameworks	49
Static Analysis, Linting, and Documentation Tools	49
Summary	50
Chapter 16: Professional Practices and Legacy Code	51
Coding Standards – Naming, Formatting, and Readability	51
Defensive Programming – Error Handling and Robustness	51
Resource Management – Cleanup and RAII-Like Patterns	51
Version Control Conventions – Git Practices for Fortran Projects . . .	51
Modernizing Legacy Code – Strategies and Tools	51
Summary	52
Conclusion: The Future of Fortran	53
What You Have Learned	53
Where Fortran Is Heading	53
Why Fortran Still Matters	53
Final Advice	53
References	54

From Fundamentals to High-Performance Scientific Computing

Fortran is more than a relic of computing history. It remains the language of choice for some of the world's most demanding scientific and engineering applications, from weather prediction and climate modeling to computational fluid dynamics and astrophysics. This book takes you from your very first Fortran program through advanced topics like object-oriented design, parallel programming with coarrays, C interoperability, numerical correctness, and high-performance optimization. You will learn not only the syntax of modern Fortran (Fortran 2018 and later) but also the reasoning behind its design, common pitfalls to avoid, and professional practices for building reliable, maintainable, and fast scientific software. Whether you are a scientist new to programming or an experienced developer stepping into the world of high-performance computing, this book provides both a structured learning path and a practical reference you can return to throughout your career.

Introduction

In November 2024, when the TOP500 list announced that three systems had reached exascale performance – capable of more than a billion billion calculations per second – the software powering those machines relied heavily on code written in Fortran [1]. This is not an accident. Over six decades after its creation, Fortran remains at the heart of high-performance computing because it was designed from the beginning for one purpose: to make numerical computation fast and reliable.

This book is about that language – not as a museum piece, but as a living, evolving tool that continues to adapt to modern hardware and programming needs. You will learn how to write Fortran code that is correct, efficient, maintainable, and portable. Along the way you will encounter topics ranging from the basics of variables and loops to advanced features like object-oriented programming, native parallelism via coarrays, seamless integration with C and other languages, and rigorous handling of floating-point arithmetic.

The central idea running through this book is simple: modern Fortran (Fortran 2003 through Fortran 2018 and beyond) is a powerful, expressive language that deserves to be understood on its own terms. It is not “old C with different syntax” or “the language your grandfather used.” While it maintains backward compatibility with decades of legacy code – a necessity for the vast scientific and engineering software ecosystems built upon it – contemporary Fortran offers features that are competitive with any modern systems or scientific programming language.

What You Will Learn

This book is organized into six parts, each building on the previous one:

Part I lays the foundations. You will learn where Fortran came from, why it survived when so many other languages faded away, and how to set up a development environment. You will write your first programs and understand the structure of Fortran source code, compilation, and execution. Then you

will dive into data types, with special attention to numerical precision – one of Fortran’s most important strengths.

Part II covers the core mechanics of programming in Fortran: control flow, loops, arrays, and input/output. Arrays are where Fortran truly shines, and this part devotes substantial attention to array syntax, multidimensional operations, and memory layout – knowledge that is essential for writing performant code.

Part III introduces modular programming. You will learn how to organize your code using procedures (functions and subroutines), interfaces, modules, and derived types. This is where Fortran becomes a language suitable for large-scale software engineering, with features like encapsulation, generic programming, and operator overloading.

Part IV explores advanced language features: dynamic memory management, object-oriented programming with type-bound procedures and polymorphism, parallel programming using Fortran’s native coarray facility, and interoperability with C and other languages. These topics equip you to build sophisticated applications that meet modern requirements for flexibility, scalability, and integration.

Part V focuses on numerical computing and performance. You will understand how floating-point arithmetic really works, how to use Fortran’s IEEE facilities to control and diagnose numerical behavior, how to write code that compilers can optimize aggressively, and how to measure and improve real-world performance on today’s hardware.

Part VI addresses professional software development practices: build systems (including the modern Fortran Package Manager), testing frameworks, profiling tools, coding standards, version control conventions, and strategies for modernizing legacy codebases. These are the skills that separate hobbyist scripts from production-quality scientific software.

How to Use This Book

If you are completely new to Fortran, read this book from beginning to end. Each chapter assumes only what came before it, and the progression is designed so that you build competence steadily without gaps. If you already know some Fortran (perhaps Fortran 77 or early Fortran 90), you may wish to skim the earlier chapters and focus on the sections covering modern features

– particularly modules, derived types, object-oriented facilities, coarrays, and interoperability.

Throughout the book you will find complete, compilable code examples. These are not pseudocode or fragments designed only to illustrate isolated syntax points. They are real programs that demonstrate how features work in practice, often building on each other across chapters to show how larger systems are constructed. Where relevant, I indicate which Fortran standard introduced a particular feature and note any differences between compilers.

You will also encounter discussions of legacy constructs – features that were common in older Fortran but are now discouraged or obsolete. Understanding these is important because you will almost certainly encounter them when working with existing codebases. I explain what they do, why they fell out of favor, and what modern alternatives you should use instead.

A Note on Standards and Compilers

The last ratified Fortran standard is ISO/IEC 1539-1:2018, commonly called Fortran 2018 [2]. Work on Fortran 2023 has been completed by the standards committee (WG5) and published as a draft international standard [3], but at the time of writing compiler support is still emerging. This book focuses primarily on Fortran 2018 as the practical “latest stable” standard, while noting where Fortran 2023 introduces interesting new features and where older standards (Fortran 2008, 2003, 95, 90) remain relevant.

The major Fortran compilers in use today are:

- GNU Fortran (`gfortran`), part of the GCC toolchain [4]
- Intel oneAPI Fortran Compiler (`ifx`, formerly `ifort`) [5]
- NVIDIA HPC SDK Fortran Compiler (`nvc`)
- Cray Fortran Compiler (now integrated into the Cray Programming Environment)
- NAG Fortran Compiler, known for strict standards conformance [6]
- LLVM Flang, an emerging open-source compiler based on LLVM infrastructure

Code in this book is written to be portable across these compilers wherever possible. When compiler-specific behavior or flags are relevant, I note them

explicitly. For concrete examples of compilation and execution, GNU Fortran is used most frequently because it is free, widely available, and representative of general practices applicable to other compilers as well.

Why Learn Fortran Today?

You might be wondering whether learning Fortran in the 2020s is a worthwhile investment. The answer depends on your goals, but here are some reasons many scientists, engineers, and HPC professionals continue to choose Fortran:

Performance. Fortran compilers are among the best in the world at generating efficient machine code for numerical computation. Decades of optimization effort focused specifically on scientific workloads mean that well-written Fortran often outperforms equivalent code in other languages, even when those languages have mature optimizing compilers.

Numerical heritage. Vast amounts of trusted, validated scientific software are written in Fortran – from weather models like the ECMWF Integrated Forecasting System to computational chemistry packages like GAMESS and Gaussian to finite-element codes used throughout engineering. If your work touches these domains, you will encounter Fortran whether you want to or not.

Modern features. Fortran 2003 introduced object-oriented programming. Fortran 2008 added native parallel programming via coarrays. Fortran 2018 incorporated improved C interoperability and additional parallel features. These are not marginal additions; they are serious, well-designed language facilities that make modern software development in Fortran practical and productive.

A growing ecosystem. The Fortran-lang community has revitalized the ecosystem in recent years with tools like the Fortran Package Manager (fpm), a standard library project (stdlib), improved documentation at fortran-lang.org [7], and active discussion forums. New projects are being started in modern Fortran, not just maintained.

Stability. The language evolves slowly and deliberately. Once you learn modern Fortran, your knowledge will remain relevant for many years. You do not need to constantly chase new syntax or framework releases.

None of this means Fortran is the right choice for every problem. For web development, rapid prototyping with rich standard libraries, or domains where ecosystem matters more than raw performance, other languages may be

better suited. But for numerical computing and high-performance scientific applications, Fortran remains an excellent – often optimal – choice.

Let us begin.

Chapter 1: The Story of Fortran — History, Evolution, and Modern Relevance

Fortran's journey from a research project at IBM in the 1950s to the backbone of modern high-performance computing is one of the most remarkable stories in the history of software. Understanding where this language came from helps explain not only why it looks the way it does but also why it has endured when so many contemporaries have faded away. This chapter traces that evolution and makes the case for Fortran's continued relevance.

The First Programming Language — FORTRAN I and the Birth of Compilers

In late 1953, John Warner Backus submitted a proposal to his superiors at IBM to develop a more practical alternative to assembly language for programming their new IBM 704 mainframe computer [8]. At the time, programs were written directly in machine code or assembly – laborious sequences of numeric opcodes and memory addresses that demanded intimate knowledge of the hardware. Backus described this process as “hand-to-hand combat with the machine,” with the machine often winning.

The 704 was chosen as the target platform not by accident. It had built-in floating-point arithmetic operations, making it especially well suited for scientific work [9]. Backus recognized that if a high-level language could be made practical for numerical computation on this machine, it would open computing to scientists and engineers who lacked the patience or training for assembly programming.

The project began in earnest during the summer of 1954. Backus assembled a team including Richard Goldberg, Sheldon F. Best, Peter Sheridan, Roy Nutt, Robert Nelson, Harlan Herrick, Robert Everett, and David Sayre [10]. They defined an initial specification in November 1954 and published a preliminary

report titled “Specifications for the IBM Mathematical FORMula TRANslating System, FORTRAN” [11]. The name itself – Formula Translator – reveals the language’s original purpose: to let scientists write programs using mathematical notation rather than machine instructions.

The first Fortran compiler was completed in 1957 and shipped with IBM 704 installations that April [12]. Skeptics had predicted that any compiler would produce code too slow to be useful – that hand-written assembly would always be faster. Backus’s team proved them wrong by implementing aggressive optimizations, including loop optimization and register allocation, making Fortran-generated code competitive with carefully written assembly in many cases [13].

FORTRAN I introduced concepts that remain fundamental to programming today:

- Variables typed as integer or real (floating-point)
- DO loops for iteration
- IF statements for conditional execution
- Subroutines and function subprograms for modularization
- Array declarations for handling collections of data
- FORMAT statements controlling input/output appearance

The language was designed specifically for numerical computation. There were no character strings, no pointers, no dynamic memory allocation – because those features were not needed for the scientific workloads Backus’s team had in mind. This focused design is one reason Fortran has remained strong in its domain: it optimized for what mattered to its users rather than trying to be everything to everyone.

Standardization and Survival – From FORTRAN 66 to Fortran 90

The success of FORTRAN I led quickly to FORTRAN II (1958), which added optional optimization levels and library functions, and FORTRAN III (never commercially released, but experimented with new features internally) [14]. By the early 1960s, Fortran was being implemented on dozens of different machines from multiple vendors, raising urgent questions about compatibility.

The answer came through standardization. In 1966, the American Standards Association (later ANSI) published the first official Fortran standard: X3.9-1966, commonly known as FORTRAN 66 [15]. This document codified what had become common practice across implementations and gave users a basis for writing portable code. The international standards body ISO adopted a compatible version in 1967 as ISO/IEC 1539:1967.

FORTRAN 77, standardized by ANSI in 1978 (X3.9-1978) and by ISO in 1979 [16], was the next major revision. It introduced several important features:

- The IF...THEN...ELSE construct for structured conditional execution
- DO loops with explicit END DO (reducing reliance on line-number-based branching)
- Character data type support
- Block DATA statements for initializing common block variables
- Improved I/O capabilities including namelist processing

FORTRAN 77 is significant not only for its features but for its longevity. It became the dominant dialect of Fortran for over two decades and remains widely used in legacy codebases today. Many scientists who learned programming in the 1980s and early 1990s learned FORTRAN 77, and the style conventions it established influenced coding practices for a generation.

The transition from “FORTRAN” (all caps, reflecting its original name as an acronym) to “Fortran” (mixed case) occurred gradually during this period and was formalized with Fortran 90. The change reflected the fact that the name had become a proper noun rather than a descriptive abbreviation – much like how “laser” is no longer written as “LASER.”

Fortran 90, published as ISO/IEC 1539:1991, was the most radical revision of the language to date [17]. It introduced features that fundamentally changed how Fortran programs could be structured:

- Free-form source code (no more rigid column-based formatting)
- Modules for encapsulation and code sharing
- Array syntax enabling whole-array operations without explicit loops
- Recursion support
- Dynamic memory allocation via the ALLOCATABLE attribute
- User-defined derived types
- Improved numerical precision control through kind parameters

Fortran 95, standardized as ISO/IEC 1539:1997, was a smaller revision that refined Fortran 90 and added a few notable features [18]:

- The WHERE construct for conditional array assignments
- Pure and elemental procedure specifications
- Support for pointer arrays
- Some performance-oriented enhancements

Together, Fortran 90 and 95 transformed the language from a procedural tool into a modern systems programming language capable of supporting large-scale software engineering practices. Code written in these standards is dramatically more readable, maintainable, and less error-prone than equivalent FORTRAN 77 code.

The Modern Renaissance – Object-Oriented Features, Parallelism, and Beyond

The twenty-first century brought Fortran into the realm of object-oriented programming and native parallelism – features that had long been associated primarily with languages like C++ and Java.

Fortran 2003 (ISO/IEC 1539-1:2004) introduced a comprehensive set of object-oriented facilities [19]:

- Type-bound procedures, allowing methods to be associated with derived types
- Polymorphism through the CLASS keyword and dynamic dispatch
- Abstract types with deferred type-bound procedures (analogous to pure virtual functions in C++)
- Type extension for inheritance hierarchies
- Finalization procedures for cleanup when objects go out of scope
- Procedure pointers for runtime selection of code behavior
- Enhanced interoperability with C through the ISO_C_BINDING module

These features did not make Fortran a clone of C++ or Java. Instead, they were designed specifically for the needs of scientific programming: type-bound procedures that operate naturally on arrays, polymorphism that works

efficiently with numerical data, and an overall design that avoids unnecessary runtime overhead.

Fortran 2008 (ISO/IEC 1539-1:2010) added native support for parallel programming through coarrays [20]. A coarray is a variable whose instances are distributed across multiple “images” (processes) running the same program. Communication between images happens through Fortran’s familiar array syntax, extended with square brackets to specify which image’s copy of the data is being accessed. This provides a high-level parallel programming model that is:

- Integrated into the language rather than requiring external libraries like MPI
- Based on single-program-multiple-data (SPMD) execution
- Capable of expressing both data parallelism and task parallelism

Fortran 2008 also strengthened other areas:

- Atomic operations for lock-free concurrent access to coarray elements
- Event-based synchronization as an alternative to global barriers
- Enhanced IEEE floating-point support aligned with the updated IEEE 754 standard
- Additional intrinsic procedures for array manipulation and numerical computation

Fortran 2018 (ISO/IEC 1539-1:2018) is the most recently ratified standard [2]. It is comparatively modest in scope but incorporates two important technical specifications:

- TS 29113: Further Interoperability of Fortran with C, adding assumed-type and assumed-rank dummy arguments, improved function pointer support, and additional C type mappings
- TS 18508: Additional Parallel Features in Fortran, introducing team-based parallelism, collective operations (`co_broadcast`, `co_sum`, etc.), and improved error handling for failed images

Work on Fortran 2023 has been completed by the standards committee [3], adding features like `TYPEOF` and `CLASSOF` type specifiers, support for varying-length character strings as a standard feature, and various refinements to existing facilities. Compiler support is gradually emerging but not yet universal.

Throughout this evolution, one principle has remained constant: backward compatibility. Every new Fortran standard is designed so that programs conforming to earlier standards continue to compile and run correctly. This is essential for the vast ecosystem of scientific software built over six decades – rewriting all of it would be impossible, and much of it works perfectly well as-is. The result is a language that can feel like multiple languages at once: a FORTRAN 77 program, a Fortran 95 array-processing script, and a Fortran 2018 object-oriented parallel application can all be valid Fortran programs.

Why Fortran Today – Supercomputers, Legacy Codebases, and New Projects

Fortran's continued relevance rests on three pillars: its dominance in high-performance computing, the sheer volume of existing code that depends on it, and a growing community building new software with modern features.

The TOP500 list, which ranks the world's most powerful supercomputers, has always been dominated by systems running Fortran-based applications [1]. The High Performance Linpack (HPL) benchmark used to generate the rankings is itself written in Fortran. Many of the application benchmarks that matter to researchers – weather models, climate simulations, computational fluid dynamics codes, molecular dynamics packages – are implemented primarily in Fortran. When you see a supercomputer announcement claiming exascale performance, the software demonstrating that performance is almost certainly calling Fortran routines at its core.

The legacy codebase argument is equally powerful. Estimates vary, but tens of millions of lines of production scientific and engineering code exist in Fortran – code that has been tested, validated, and trusted over decades of real-world use. This code computes everything from aircraft stress analysis to financial risk models to seismic survey interpretations. It will not be rewritten in another language any time soon because:

- The algorithms are correct and well understood
- Rewriting introduces the risk of subtle bugs
- The performance is already excellent
- The cost would be enormous

This legacy code needs maintenance, extension, and integration with modern software – work that requires Fortran knowledge. Engineers who can read, understand, and modify existing Fortran code are in steady demand across aerospace, defense, energy, finance, and research institutions.

But Fortran is not only about maintaining the past. The ecosystem has seen genuine revitalization in recent years:

- The Fortran-lang community (fortran-lang.org) was formed in late 2019 to coordinate modernization efforts [7]
- The Fortran Package Manager (fpm) provides a modern build system and dependency management tool comparable to Cargo for Rust or pip for Python [21]
- A standard library project (stdlib.fortran-lang.org) is under development to provide commonly needed utilities
- New compilers like LLVM Flang bring fresh implementation approaches and better integration with modern toolchains
- Projects in machine learning, quantum computing, and other emerging fields are choosing modern Fortran for performance-critical components

The arXiv paper “Fortran... ok, and what’s next?” (2024) captures this renewed energy, noting that “modern Fortran is a standardized language that includes object-oriented and parallel programming paradigms” and highlighting the active community working to improve the ecosystem [22].

How This Book Is Organized – A Roadmap for the Reader

With this historical context established, let me outline how the rest of this book is structured so you can navigate it effectively.

Part I (Chapters 2-3) gets you started with practical Fortran programming: writing programs, compiling them, understanding variables and data types, and getting comfortable with numerical precision. By the end of Chapter 3 you will be able to write correct Fortran programs that handle numbers reliably.

Part II (Chapters 4-6) covers the core language mechanics: control flow (if statements, loops), arrays (Fortran’s superpower), and input/output. These chapters establish the foundation for real computational work.

Part III (Chapters 7-9) introduces modular programming practices: procedures with proper interfaces, modules for code organization, derived types, and object-oriented features. This is where you learn to write Fortran suitable for large projects.

Part IV (Chapters 10-12) tackles advanced topics: dynamic memory management, parallel programming with coarrays, and interoperability with C and other languages. These chapters prepare you for modern high-performance applications.

Part V (Chapters 13-14) focuses on numerical correctness and performance optimization – understanding floating-point behavior, writing code that compilers can optimize aggressively, and measuring real-world performance.

Part VI (Chapters 15-16) addresses professional practices: build systems, testing, profiling, coding standards, version control, and modernizing legacy code. These are the skills that separate production-quality software from research prototypes.

Chapter 2: Your First Fortran Programs

— Structure, Compilation, Execution

The best way to learn any programming language is to write programs in it. This chapter gets you started immediately: you will set up a development environment, understand the basic structure of a Fortran source file, write and compile your first programs, and learn how the compilation process works. By the end of this chapter you will have a working Fortran toolchain and the confidence to experiment with the language.

The Anatomy of a Fortran Program — Source Files and Program Units

A Fortran program is composed of one or more source files, each containing one or more “program units.” Understanding these building blocks is essential because they determine how your code is organized and compiled.

The main types of program units are:

- **Main program:** The entry point of execution, declared with a PROGRAM statement. Every complete Fortran application must have exactly one main program.
- **Module:** A container for declarations (types, constants, interfaces) that can be shared across multiple program units via USE statements. Modules are the backbone of modern Fortran code organization.
- **Subroutine:** A procedure that performs actions but does not return a value through its name. Called using CALL statement.
- **Function:** A procedure that computes and returns a single value through its name. Used in expressions like mathematical functions.
- **Submodule** (Fortran 2008+): A refinement of modules that separates interface declarations from implementation details, reducing recompilation dependencies.

A minimal Fortran program consists of just a main program unit. Let us look at the simplest possible example:

```
1 program hello
2   implicit none
3   print *, "Hello, world!"
4 end program hello
```

This program does exactly what you expect: it prints “Hello, world!” to the standard output and terminates. Let me break down each line:

- `program hello` declares the beginning of a main program named “hello”. The name is optional but recommended for clarity.
- `implicit none` is one of the most important statements in modern Fortran. It disables implicit typing, meaning every variable must be explicitly declared before use. Without this statement, Fortran applies legacy rules where variables starting with I through N are implicitly integers and all others are implicitly real numbers – a source of countless bugs. Always include `implicit none` at the start of every program unit.
- `print *, "Hello, world!"` outputs the string to standard output. The asterisk (*) is a shorthand for “use default formatting.”
- `end program hello` marks the end of the main program. Including the name here (matching the opening statement) is good practice that helps with code navigation and maintenance.

Fortran source files typically use one of several extensions:

- `.f90` or `.f95`: Free-form source (modern standard, lines are not position-sensitive)
- `.f03`, `.f08`, `.f18`: Also free-form, indicating the target standard version
- `.f` or `.for`: Fixed-form source (legacy format with rigid column rules, rarely used for new code)

For all new code, use free-form source (`.f90` is the most common extension). Free-form source allows you to write on any column of the line, use meaningful line lengths, and include comments starting with `!` anywhere. Fixed-form source requires statements to begin in column 7 or later, uses column 1 for labels or continuation markers, and restricts comments to lines starting with `C`, `c`, or `*` in column 1 – constraints that served punch cards well but serve modern programmers poorly.

Setting Up Your Environment – Compilers and Tools

To compile and run Fortran programs you need a Fortran compiler. The most widely available option is GNU Fortran (`gfortran`), part of the GCC (GNU Compiler Collection) toolchain. Here are instructions for common platforms:

Linux: Most distributions include `gfortran` in their package repositories:

- Ubuntu/Debian: `sudo apt install gfortran`
- Fedora/RHEL: `sudo dnf install gcc-gfortran`
- Arch Linux: `sudo pacman -S gcc-fortran`

macOS: Using Homebrew: `brew install gcc` (this installs `gfortran` as part of the GCC package). Note that on macOS the executable may be named `gfortran-14` or similar, depending on the version.

Windows: Options include:

- Installing MinGW-w64 with GCC/GFortran via MSYS2
- Using the LLVM Flang compiler (available via `vcpkg` or direct download)
- Intel oneAPI HPC Toolkit (free for individual developers)
- Cygwin with `gfortran` package

After installation, verify your compiler is working by running:

```
1 gfortran --version
```

You should see output indicating the version number and build information.

Beyond the compiler itself, a few additional tools enhance your Fortran development experience:

- **fpm (Fortran Package Manager):** A modern build system and package manager that simplifies project management, dependency handling, and testing [21]. Install from <https://github.com/fortran-lang/fpm> following their instructions.
- **Text editor or IDE:** Any editor works, but editors with Fortran syntax highlighting improve readability. Options include VS Code (with the Modern Fortran extension), Vim/Neovim (with a Fortran plugin), Emacs (with `f90-mode`), and dedicated IDEs like Visual Studio (with Intel or NVIDIA compiler integration).

- **Version control:** Git is the standard for source code management and integrates well with all Fortran toolchains.

For this chapter's examples, I assume you have gfortran installed and accessible from your command line. The concepts apply equally to other compilers, though specific commands and flags may differ.

Hello World and Beyond – Free-Form Source and Basic Structure

Let us write our first program properly. Create a file named `hello.f90` with the following content:

```
1  ! hello.f90 - First Fortran program
2  program hello
3      implicit none
4      print *, "Hello, world!"
5  end program hello
```

The line beginning with `!` is a comment – text ignored by the compiler but useful for human readers. In free-form source, `!` starts a comment that extends to the end of the line. You can also place inline comments after code on the same line:

```
1  print *, "Hello, world!" ! Print greeting to standard output
```

Now compile and run the program:

```
1  gfortran -o hello hello.f90
2  ./hello
```

The first command compiles `hello.f90` and produces an executable named `hello` (on Windows this would be `hello.exe`). The second command runs it. You should see:

```
1 Hello, world!
```

Let us expand this to a slightly more interesting example that demonstrates variable declaration and simple computation:

```
1  ! temperature.f90 - Convert Celsius to Fahrenheit
2  program temperature
3      implicit none
4      real :: celsius
5      real :: fahrenheit
6
7      ! Read temperature in Celsius from user
8      print *, "Enter temperature in Celsius:"
9      read *, celsius
10
11     ! Convert to Fahrenheit
12     fahrenheit = 9.0 / 5.0 * celsius + 32.0
13
14     ! Display result
15     print *, "Temperature in Fahrenheit:", fahrenheit
16 end program temperature
```

This program introduces several important concepts:

- **Variable declaration:** `real :: celsius` declares a variable named `celsius` of type `real` (floating-point). The double colon (`::`) separates the type specification from the variable list. Multiple variables of the same type can be declared on one line: `real :: x, y, z`.
- **Input:** `read *, celsius` reads a value from standard input and stores it in the variable `celsius`. Like `print`, the asterisk means “use default formatting.”
- **Arithmetic expression:** The conversion formula uses standard mathematical operators (`/`, `*`, `+`). Note that `9.0 / 5.0` uses real literals (with decimal points), ensuring floating-point division rather than integer division.

Compile and run this program:

```
1 gfortran -o temperature temperature.f90
2 ./temperature
```

When prompted, enter a Celsius temperature and observe the Fahrenheit result.

A few important observations about Fortran syntax:

- The language is case-insensitive: `PROGRAM`, `Program`, and `program` are equivalent. However, convention favors lowercase for keywords and mixed or lowercase for identifiers. Consistency matters more than any particular choice.
- Statement labels (numbers at the beginning of lines used with `GOTO` statements) are a legacy feature that modern code generally avoids. You will see them in older programs but should not use them in new code.
- Fortran ignores extra whitespace between tokens, so you can format your code for readability without affecting its meaning.

Compilation and Linking – From Source to Executable

Understanding how Fortran code becomes a running program helps you diagnose build errors and organize larger projects. The process involves two main stages: compilation and linking.

Compilation translates each source file into an object file containing machine code for that file's program units, along with information about what symbols (variables, procedures) it defines and what it needs from other files. With `gfortran`:

```
1 gfortran -c hello.f90
```

This produces `hello.o` (or `hello.obj` on Windows). The `-c` flag tells the compiler to compile only, not to link.

Linking combines one or more object files into a single executable, resolving references between them. If your program spans multiple source files:

```
1 gfortran -c main.f90 utils.f90
2 gfortran -o myprogram main.o utils.o
```

Or in one step (the compiler handles the intermediate object files automatically):

```
1 gfortran -o myprogram main.f90 utils.f90
```

When you use modules, the compilation process has an additional step: compiling a module produces both an object file and a “module file” (with extension `.mod` for `gfortran`) containing the interface information that other program units need. For example:

```
1 gfortran -c mymodule.f90      # Produces mymodule.o and mymodule.mod
2 gfortran -o myprogram main.f90 mymodule.o # main.f90 uses mymodule
```

The module file must be available when compiling any source that USEs the module. Build systems like `fpm` or `CMake` handle these dependencies automatically, but understanding them helps when troubleshooting compilation errors.

Useful compiler flags for development:

- `-Wall`: Enable most warnings (catches common mistakes)
- `-Wextra`: Additional warnings beyond `-Wall`
- `-pedantic`: Warn about non-standard extensions
- `-g`: Include debugging information for use with debuggers like `gdb`
- `-O0`: No optimization (faster compilation, easier debugging)
- `-O2` or `-O3`: Enable optimizations (use for production builds)

A good development command is:

```
1 gfortran -Wall -Wextra -pedantic -g -O0 -o myprogram main.f90
```

For production builds:

```
1 gfortran -Wall -O3 -o myprogram main.f90
```

Common First Errors and How to Fix Them

Every programmer encounters errors. Fortran compilers generally produce clear error messages, but understanding common pitfalls helps you resolve issues quickly.

Error 1: Implicit typing violations

If you forget `implicit none` and use an undeclared variable, Fortran silently assigns it a type based on its first letter. This often leads to subtle bugs – for example, treating what you intended as a real number as an integer because its name starts with “I.” Always include `implicit none` to catch these mistakes at compile time.

Error 2: Type mismatches in I/O

```
1 integer :: count = 42
2 print *, count      ! Works fine
3 read *, count       ! If user types "3.14", behavior is implementation-defined
```

When reading with list-directed I/O (*), the compiler attempts to match the input format to the variable type. Providing a real number when an integer is expected may truncate the value, produce a warning, or cause an error depending on the compiler and settings. Be explicit about expected formats in production code.

Error 3: Integer division

```
1 real :: result
2 result = 1 / 3      ! Result is 0.0 (integer division), not 0.333...
3 result = 1.0 / 3.0 ! Correct: floating-point division gives 0.333...
```

This is one of the most common mistakes for beginners. In Fortran, dividing two integers produces an integer result (truncated toward zero). To get floating-point division, at least one operand must be real. Write `1.0 / 3.0`, `real(1) / 3`, or `1 / 3.0` – any form that ensures at least one operand is real.

Error 4: Missing END statements

```

1 program example
2   implicit none
3   print *, "Hello"
4   ! Forgot: end program example

```

Forgetting an END statement produces a clear compiler error about unexpected end of file. Modern editors with syntax highlighting and matching-bracket features help catch these mistakes visually.

Error 5: Module dependency order

If `main.f90` USEs a module defined in `mymodule.f90`, you must compile the module first (or list it before `main.f90` on the command line):

```

1 gfortran -o myprogram mymodule.f90 main.f90 # Correct order
2 gfortran -o myprogram main.f90 mymodule.f90 # May fail: module not yet compiled

```

Build systems handle this automatically, but when compiling manually you need to respect the dependency order.

Error 6: Syntax errors from legacy habits

Fortran tolerates some flexibility that can mask mistakes:

- Missing commas in argument lists sometimes work due to implicit continuation rules
- Trailing spaces or inconsistent indentation do not affect compilation but hurt readability
- Using `DOUBLE PRECISION` instead of the modern `REAL(KIND=...)` approach works but is non-portable

Develop good habits early. Use modern syntax, be explicit about types and interfaces, and let compiler warnings guide you toward cleaner code.

A Slightly Larger Example – Organizing Code from the Start

Let us look at a small but complete example that demonstrates good practices from the beginning: separating concerns into modules and keeping the main program focused on orchestration rather than implementation details.

File `constants.f90`:

```

1  ! constants.f90 - Physical constants module
2  module constants
3      implicit none
4      public :: pi, gravitational_constant
5
6      ! Mathematical constant pi to high precision
7      real(kind=8), parameter :: pi = 3.141592653589793d0
8
9      ! Gravitational constant in m^3 kg^-1 s^-2
10     real(kind=8), parameter :: gravitational_constant = 6.67430e-11_8
11 end module constants

```

File geometry.f90:

```

1  ! geometry.f90 - Geometric calculation procedures
2  module geometry
3      use constants, only: pi
4      implicit none
5  contains
6
7      !> Compute the area of a circle given its radius
8      function circle_area(radius) result(area)
9          real(kind=8), intent(in) :: radius
10         real(kind=8) :: area
11
12         if (radius < 0.0_8) then
13             area = -1.0_8 ! Signal error with negative value
14         else
15             area = pi * radius**2
16         end if
17     end function circle_area
18
19     !> Compute the volume of a sphere given its radius
20     function sphere_volume(radius) result(volume)
21         real(kind=8), intent(in) :: radius
22         real(kind=8) :: volume
23
24         if (radius < 0.0_8) then
25             volume = -1.0_8
26         else
27             volume = 4.0_8 / 3.0_8 * pi * radius**3
28         end if
29     end function sphere_volume
30
31 end module geometry

```

File main.f90:

```

1  ! main.f90 - Demonstration of geometry calculations
2  program geometry_demo
3      use geometry, only: circle_area, sphere_volume
4      implicit none
5      real(kind=8) :: radius
6      real(kind=8) :: area, volume
7
8      print *, "Enter radius:"
9      read *, radius
10
11     area = circle_area(radius)
12     volume = sphere_volume(radius)
13
14     if (area >= 0.0_8) then
15         print '(A, F12.4)', "Circle area:      ", area
16         print '(A, F12.4)', "Sphere volume:   ", volume
17     else
18         print *, "Error: radius must be non-negative"
19     end if
20
21 end program geometry_demo

```

To build and run this multi-file project:

```

1  gfortran -Wall -Wextra -o geometry_demo constants.f90 geometry.f90 main.f90
2  ./geometry_demo

```

This example demonstrates several important practices:

- **Modules organize related functionality:** Constants live in one module, calculation procedures in another.
- **Explicit interfaces:** Functions are defined inside modules, giving them explicit interfaces that the compiler can check at call sites.
- **Error handling:** Negative radius values are detected and reported rather than producing meaningless results.
- **Formatted output:** The print statements use format strings ('(A, F12.4)') for controlled numeric display rather than relying on default formatting.

You will learn more about each of these features in upcoming chapters. For now, the key takeaway is that even small programs benefit from thoughtful organization – and Fortran provides the tools to do it well from the very beginning.

Chapter 3: Data, Types, and Numbers

— Variables, Kinds, Precision, Literals

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Declaring Variables — Types, Names, and Style

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

The Intrinsic Data Types — Integer, Real, Complex, Logical, Character

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Kinds and Numerical Precision — Choosing the Right Representation

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Literals, Constants, and Parameters — Exact Values in Code

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Operators and Expressions – Arithmetic, Relational, Logical

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Chapter 4: Control Flow and Iteration

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Conditional Execution — IF Constructs and SELECT CASE

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Looping with DO — Count-Controlled Repetition

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Advanced Loop Patterns — WHILE-like Loops, Exit, Cycle, Nested Loops

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Structured Programming Discipline — Avoiding GOTO

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Algorithmic Examples — Search, Sort, and Iterative Methods

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Chapter 5: Arrays and Array Operations

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Declaring Arrays – Shape, Bounds, and Initialization

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Multidimensional Arrays – Matrices and Beyond

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Array Syntax and Whole-Array Operations – Vectorized Code

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Slices, Sections, and Masked Arrays – Selective Access

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Memory Layout and Performance – Column-Major Order and Cache Behavior

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Chapter 6: Input and Output

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

List-Directed I/O — Simple Reading and Writing

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Formatted I/O — Edit Descriptors and Precision Control

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Internal Files — String Formatting and Parsing

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Unformatted and Stream I/O — Binary Data and Portability

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Robust Input Handling — Error Checking and User Interaction

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Chapter 7: Procedures and Interfaces

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Functions and Subroutines – Designing Reusable Procedures

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Argument Passing and Intent Attributes – Clarity and Safety

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Optional and Keyword Arguments – Flexible Interfaces

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Explicit Interfaces and Interface Blocks – Compile-Time Checking

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Generic Interfaces and Operator Overloading – Polymorphic Names

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Chapter 8: Modules and Program Organization

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

What Modules Are For — Encapsulation and Code Sharing

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Module Structure — Public, Private, and USE Associations

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Submodules — Separating Interface from Implementation

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Scope Rules and Name Resolution — Avoiding Collisions

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Organizing Large Projects – Module Architecture Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Chapter 9: Derived Types and Object-Oriented Fortran

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Defining Derived Types — Structuring Complex Data

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Type-Bound Procedures — Methods in Fortran

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Polymorphism and the CLASS Keyword — Dynamic Dispatch

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Abstract Types and Type Extension — Inheritance Hierarchies

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Design Patterns with Modern Types – When OOP Helps

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Chapter 10: Memory Management — Allocatables, Pointers, and Dynamic Data

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Allocatable Variables — Dynamic Sizing and Lifecycle

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Allocation Patterns — Arrays, Derived Types, and Error Handling

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Pointers and Pointer Semantics — When They Make Sense

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Common Memory Pitfalls — Leaks, Dangling References, Double Free

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Modern Alternatives — Preferring Allocatables Over Pointers

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Chapter 11: Parallel Programming with Coarrays

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

The Coarray Model – Images, Teams, and Shared Variables

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Declaring and Using Coarrays – Basic Syntax

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Synchronization – Event-Based Coordination with SYNC

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Communication Patterns – Scatter, Gather, Reductions

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Practical Parallel Programs – Real Examples and Pitfalls

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Chapter 12: Interoperability — Connecting Fortran to the World

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Why Interoperate — Ecosystems and Libraries

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

The ISO_C_BINDING Module — Types, Constants, and Procedures

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Calling C from Fortran — Practical Examples

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Calling Fortran from C — Exporting Interfaces

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Beyond C — Python, Julia, and Other Language Bridges

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Chapter 13: Numerical Correctness and IEEE Floating-Point Facilities

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

How Floating-Point Works — Representation and Limitations

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Precision Selection and Kind Parameters — Getting It Right

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

IEEE Arithmetic Facilities — Querying and Controlling Behavior

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Numerical Stability — Algorithms That Work in Practice

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Reproducibility – Making Results Repeatable

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Chapter 14: Performance Optimization and High-Performance Computing

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Understanding Compiler Optimizations – Levels and Flags

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Writing Vectorizable Code – What Compilers Can Accelerate

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Memory Access Patterns – Locality, Stride, and Cache Efficiency

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Profiling and Benchmarking – Measuring Real Performance

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

HPC Considerations — Scaling to Large Systems

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Chapter 15: Build Systems, Testing, and Modern Toolchains

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

The Fortran Package Manager (fpm) – Modern Builds Made Simple

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

CMake for Fortran Projects – Cross-Platform Build Automation

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Traditional Makefiles – When They Still Make Sense

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Testing Strategies – Unit Tests and Validation Frameworks

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Static Analysis, Linting, and Documentation Tools

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Chapter 16: Professional Practices and Legacy Code

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Coding Standards — Naming, Formatting, and Readability

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Defensive Programming — Error Handling and Robustness

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Resource Management — Cleanup and RAI-Like Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Version Control Conventions — Git Practices for Fortran Projects

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Modernizing Legacy Code — Strategies and Tools

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Conclusion: The Future of Fortran

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

What You Have Learned

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Where Fortran Is Heading

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Why Fortran Still Matters

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

Final Advice

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.

References

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/modernfortranprogramming>.