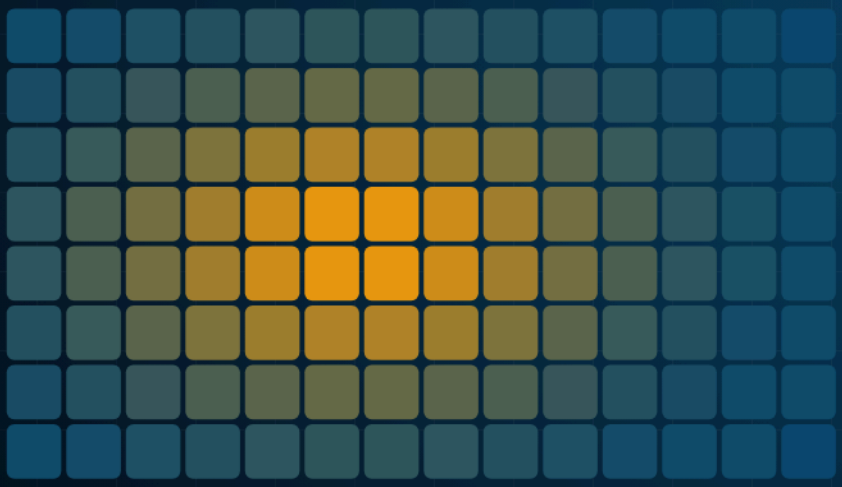


```
do concurrent (i = 2:n-1)
  t_new(i) = t(i) + r * &
    (t(i-1) - 2*t(i)
      + t(i+1))
end do

real(dp) :: t(n)
```

SCIENTIFIC COMPUTING

Modern Fortran for Scientists & Engineers

From your first program to numerical methods and parallel computing

Arrays, modules, numerical methods, OpenMP, and a real model — with exercises and solutions in every chapter

ARRAYS · MODULES · NUMERICS · OPENMP

THENEXBOOKS

Modern Fortran for Scientists and Engineers

From Your First Program to Numerical Methods and Parallel Computing

Arrays, Modules, Derived Types, Numerical Methods, OpenMP, and Real Projects

First Edition

THENEXBOOKS

Modern Fortran for Scientists and Engineers
From Your First Program to Numerical Methods and Parallel Computing

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples and exercise solutions may be used freely for learning and in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Examples follow the Fortran 2008 and 2018 standards; compiler support for individual features varies, so always consult your compiler's documentation.

Intel and oneAPI are trademarks of Intel Corporation. LLVM and other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement or affiliation.

Riverbend Environmental Lab, the Alder River, its monitoring stations, the people in the examples, and all sample data are fictional. Any resemblance to real organizations, people, or measurements is coincidental.

Preface

Fortran is one of the oldest programming languages still in daily use, and also one of the most misunderstood. Many people picture punched cards and capital letters. Modern Fortran is something quite different: a clean, strongly typed language with whole-array arithmetic, modules, derived types, object-oriented features, built-in parallelism, and some of the best optimizing compilers available. Weather and climate models, engineering simulations, and physics codes around the world are written in it, and they need people who can read, maintain, and extend them.

This book teaches Fortran the modern way from the first page. We start with a simple program and build up, one idea at a time, to modules, derived types, numerical methods, performance tuning, and parallel computing. Along the way you will learn not only the language but the habits that make scientific code trustworthy: choosing precision deliberately, checking errors, testing numerical results, and keeping code organized as it grows.

Every chapter ends with graded exercises and worked solutions. By the end of the book you will have built a complete river-flow model as a real Fortran project — reading measured data, solving differential equations, comparing simulation with observations, and running in parallel — and you will be ready to work on Fortran code of your own or someone else's.

Who This Book Is For

- **Students and researchers** in science and engineering who need to write fast numerical code.
- **Programmers joining a project** with an existing Fortran code base, modern or legacy.
- **Python, MATLAB, and C users** who want to move performance-critical work into a compiled language.
- **Anyone curious** about the language that still powers much of scientific computing.

What You Will Learn

- How to install a compiler, build programs, and manage projects with the Fortran Package Manager.
- Types, kinds, and floating-point precision — and the numerical pitfalls to avoid.
- Arrays in depth: sections, masks, reductions, and dynamic allocation.
- Procedures, modules, derived types, object-oriented and generic programming.
- File input and output, error handling, testing, debugging, and interoperability with C and Python.
- Numerical methods, performance optimization, parallel computing, and modernizing legacy code.

How This Book Is Organized

Part I — Getting Started (Chapters 1–5) covers setup, types and kinds, expressions, input and output, and control flow. **Part II — Arrays and Numerical Building Blocks** (Chapters 6–10) covers arrays, dynamic memory, strings, and floating-point arithmetic. **Part III — Structuring Programs** (Chapters 11–15) covers procedures, arguments, modules, derived types, and files. **Part IV — Modern Fortran Features** (Chapters 16–20) covers object-oriented and generic programming, error handling, interoperability, and testing. **Part V — Numerical Methods in Practice** (Chapters 21–25) covers root finding, integration, linear algebra, differential equations, grid simulations, and performance. **Part VI — Parallel Computing, Legacy Code, and Case Study** (Chapters 26–28) brings everything together. The appendices provide a cheat sheet, templates and reference tables, a checklist, and a glossary.

The Running Example: Riverbend Environmental Lab

Throughout the book you join **Riverbend Environmental Lab**, a fictional university research group that monitors the Alder River with four measuring stations. Rainfall, water levels, flow, and temperature readings give every chapter real work to do, and in the final chapter the pieces come together as a complete rainfall–runoff model.

Conventions

Code appears in `monospace`. Examples use free-form source in `.f90` files, `implicit none`, and double precision through a `dp` kind parameter. File captions such as `src/riverbend_stats.f90` show where code belongs. Exercises are labeled **[Easy]**, **[Medium]**, or **[Challenge]**.

> **TIP**
> A recommended practice, or a "Try It" prompt to pause and experiment.

> **NOTE**
> Background information or a useful aside.

> **WARNING**
> A common mistake that causes wrong results or crashes.

Contents

Preface

PART I — GETTING STARTED

1 Why Fortran Today: Where It Shines in Science and Engineering

2 Setting Up: Compilers, the Fortran Package Manager, and Your First Program

3 Variables, Types, and Kinds: Integers, Reals, and Precision

4 Expressions, Intrinsic Functions, and Basic Input and Output

5 Control Flow: if, select case, and do Loops

PART II — ARRAYS AND NUMERICAL BUILDING BLOCKS

6 Arrays: Declaration, Indexing, and Whole-Array Operations

7 Array Sections, Constructors, Masks, and Reductions

8 Allocatable Arrays and Dynamic Memory

9 Character Strings and Text Processing

10 Floating-Point Arithmetic: Precision, Rounding, and Numerical Pitfalls

PART III — STRUCTURING PROGRAMS

11 Procedures: Functions and Subroutines

12 Arguments in Depth: intent, optional, Assumed-Shape Arrays, pure and elemental

13 Modules: Organizing Code, Visibility, and Explicit Interfaces

14 Derived Types: Building Your Own Data Structures

15 File Input and Output: Formatted, Unformatted, and Stream Access

PART IV — MODERN FORTRAN FEATURES

16 Object-Oriented Fortran: Type-Bound Procedures, Extension, and Polymorphism

17 Generic Programming: Generic Interfaces, Operator Overloading, and Abstract Interfaces

18 Robust Programs: Error Handling with iostat, stat, and error stop

19 Interoperability: Calling C from Fortran and Fortran from C and Python

20 Testing, Debugging, and Tooling: Compiler Checks, Unit Tests, and fpm

PART V — NUMERICAL METHODS IN PRACTICE

21 Root Finding and Numerical Integration

22 Linear Algebra: Solving Systems and Using BLAS and LAPACK

23 Ordinary Differential Equations: Euler and Runge–Kutta Methods

24 Simulations on Grids: Finite Differences and Diffusion

25 Performance: Memory Layout, Profiling, and Optimization

PART VI — PARALLEL COMPUTING, LEGACY CODE, AND CASE STUDY

26 Parallel Fortran: do concurrent, OpenMP, and an Introduction to Coarrays

27 Working with Legacy Code: From FORTRAN 77 to Modern Fortran

28 Case Study: Building the Riverbend River-Flow Model

APPENDICES

A Modern Fortran Cheat Sheet

B Templates and Reference Tables

C Code Review and Numerical Checklist

D Glossary

PART I

Getting Started

Why Fortran, installing compilers and fpm, types and kinds, expressions and basic I/O, and control flow.

1 · Why Fortran Today

2 · Setting Up

3 · Variables, Types, and Kinds

4 · Expressions, Intrinsic Functions, and Basic Input and Output

5 · Control Flow

CHAPTER 1 · PART I — GETTING STARTED

Why Fortran Today: Where It Shines in Science and Engineering

Fortran is one of the oldest programming languages still in daily use, and it is also a modern, actively standardized language built for fast numerical work. In this chapter we look at what Fortran does well, how it grew from FORTRAN 77 into the language we use today, when another language is the better choice, and how this book is organized around a realistic research lab.

A Language Built for Numbers

The name Fortran comes from "Formula Translation." From the very beginning the language had one clear purpose: let scientists and engineers write mathematical formulas in a form close to the way they write them on paper, and let a compiler turn those formulas into fast machine code. That purpose has never changed. Every revision of the language since then has added features, but the center of gravity has stayed the same: numbers, arrays of numbers, and the loops and formulas that transform them.

This focus matters because most programming languages are general-purpose tools that later acquired numerical libraries. Fortran went the other way. Multidimensional arrays, complex numbers, a rich set of mathematical intrinsic functions, and precise control over floating-point precision are part of the core language rather than add-ons. When you write a Fortran program that adds two matrices, the compiler knows that the operands are arrays of a known type and rank, and it can generate tight, vectorized code without any help from you.

Array math as a first-class idea

The most visible strength of modern Fortran is whole-array arithmetic. Suppose we have six hourly river-level readings and want the change from one hour to the next. In many languages you write a loop; in Fortran you can write the formula directly on array sections:

level_rise.f90

```
program level_rise
   use, intrinsic :: iso_fortran_env, only: real64
   implicit none
   integer, parameter :: dp = real64
   real(dp) :: level_m(6), rise_m(5)

   level_m = [1.10_dp, 1.11_dp, 1.15_dp, 1.24_dp, 1.33_dp, 1.38_dp]
   rise_m = level_m(2:6) - level_m(1:5)
   print '(a, 5f6.2)', 'Hourly rise (m):', rise_m
end program level_rise
```

Running it prints:

```
Hourly rise (m):  0.01  0.04  0.09  0.09  0.05
```

Do not worry about every detail yet. The line `rise_m = level_m(2:6) - level_m(1:5)` subtracts elements 1 to 5 from elements 2 to 6, one pair at a time, and stores the five differences. There is no explicit loop, no index variable to get wrong, and the compiler is free to process the elements in whatever order runs fastest. We meet array sections properly in Chapter 6 and Chapter 7, and the `_dp` suffix and `iso_fortran_env` in Chapter 3.

Numerical performance

Fortran has a reputation for speed, and the reputation has a technical basis rather than a mystical one. Three properties of the language make it easy for compilers to optimize:

- **Arrays are part of the type system.** The compiler knows an array's rank, element type, and often its shape, so it can plan memory access and use vector instructions.
- **Aliasing rules are strict.** By default, Fortran assumes that two different dummy arguments of a procedure do not overlap in memory unless you say otherwise. In C, the compiler must usually assume that two pointers might refer to the same memory, which blocks many optimizations unless the programmer adds hints.
- **The language is deliberately small.** There is no hidden object machinery or garbage collector running behind arithmetic. What you write is close to what runs.

None of this means a Fortran program is automatically fast. A poorly organized loop can be slow in any language, and Chapter 25 is devoted to memory layout and profiling. But Fortran gives a careful programmer a short path from formula to efficient machine code, and it gives the compiler fewer reasons to be cautious.

Long-lived scientific codes

Scientific software often outlives the people who started it. Weather and climate models, structural analysis packages, computational chemistry codes, and reactor physics programs are maintained for decades by successive generations of researchers. Fortran supports this kind of longevity unusually well because the standards committee takes backward compatibility seriously. Code written to an older standard usually still compiles today, sometimes with warnings. That stability is valuable when a lab's results depend on a model that has been validated over many years.

The flip side is that many programmers first meet Fortran through old code: uppercase keywords, fixed columns, numbered statements, and `GOTO`. That style is not what this book teaches, but you will learn to read it, and Chapter 27 shows how to modernize it step by step.

High-performance computing

Fortran remains a common language on supercomputers and research clusters. It has standard features for parallel work (`do concurrent` and coarrays), it works with the widely used OpenMP directives and MPI libraries, and decades of numerical libraries such as BLAS and LAPACK were written in or for it. When you build a simulation that must run for hours on many cores, Fortran is a natural fit. We introduce parallel Fortran in Chapter 26 and use BLAS and LAPACK in Chapter 22.

> **NOTE**
>
> You will see both spellings, FORTRAN and Fortran. The all-capitals form is traditional for the early versions up to FORTRAN 77. From Fortran 90 onward, the official name is written with only the first letter capitalized. In this book "Fortran" means modern Fortran unless we say otherwise.

How the Language Evolved

Fortran is defined by international standards. Compiler vendors implement a standard, so a program written to the standard can be compiled by any conforming compiler. Knowing the major standards helps you read older code, understand compiler documentation, and choose the right flags.

Standard	What it brought (highlights)
FORTRAN 77	Structured `IF … THEN … ELSE`, `CHARACTER` type, standardized I/O. Fixed-form source with column rules.
Fortran 90	Free-form source, modules, whole-array operations, allocatable arrays, derived types, kind parameters, recursion, `implicit none`.
Fortran 95	Mostly refinements: `pure` and `elemental` procedures, default initialization of derived-type components.
Fortran 2003	Object-oriented features (type extension, type-bound procedures), C interoperability, IEEE floating-point modules, stream I/O.
Fortran 2008	Coarrays for parallel programming, `do concurrent`, submodules, the `block` construct.
Fortran 2018	Extended coarray features (teams, collective subroutines such as `co_sum`), further C interoperability, many smaller clarifications.
Fortran 2023	A smaller revision with conveniences such as conditional expressions; supported in recent compiler versions.

The biggest single jump was from FORTRAN 77 to Fortran 90. Almost everything that makes Fortran pleasant today, from free-form source to modules and array syntax, arrived in that standard. Fortran 2003 and Fortran 2008 then turned the language into one that can express large, well-structured

programs with objects, generic interfaces, and parallelism. This book targets Fortran 2008 and Fortran 2018 features, which all major compilers support well, and we mention Fortran 2023 only in small notes.

Legacy Style and Modern Style, Side by Side

The fastest way to see how far the language has moved is to write the same small task in both styles. The task: compute the total and mean rainfall for six hourly readings from the Mill Bridge station. First, the way it might look in an old FORTRAN 77 program:

rain77.f (legacy fixed form)

```
C     TOTAL AND MEAN RAINFALL - FIXED FORM
      PROGRAM RAIN
      REAL R(6), TOTAL
      INTEGER I
      DATA R /0.0, 0.4, 2.6, 5.2, 3.1, 0.8/
      TOTAL = 0.0
      DO 10 I = 1, 6
         TOTAL = TOTAL + R(I)
   10 CONTINUE
      WRITE (*, 100) TOTAL, TOTAL / 6.0
  100 FORMAT ('TOTAL =', F8.3, '  MEAN =', F8.3)
      END
```

Several things stand out. A `C` in column 1 marks a comment. Statements must begin in column 7 or later, and labels such as `10` and `100` live in columns 1 to 5. The loop ends at a labeled `CONTINUE` statement. There is no `IMPLICIT NONE`, so if you mistype `TOTAL` as `TOTL` the compiler silently invents a new variable. The type `REAL` is whatever the compiler's default real happens to be, typically about seven significant digits.

Now the same task in the style this book teaches:

rain_summary.f90

```
program rain_summary
   use, intrinsic :: iso_fortran_env, only: real64
   implicit none
   integer, parameter :: dp = real64
   real(dp), parameter :: rain_mm(6) = &
      [0.0_dp, 0.4_dp, 2.6_dp, 5.2_dp, 3.1_dp, 0.8_dp]
   real(dp) :: total

   total = sum(rain_mm)
   print '(a, f8.3, a, f8.3)', 'Total (mm) =', total, &
      '   Mean (mm) =', total / size(rain_mm)
end program rain_summary
```

```
Total (mm) =  12.100   Mean (mm) =   2.017
```

The modern version uses free-form source (no column rules), lowercase keywords, and `implicit none`, which forces every variable to be declared. The precision is explicit: `real64` is a 64-bit real with roughly 15 to 16 significant digits. The data is a named constant array, and the intrinsic functions `sum` and `size` replace the hand-written loop and the hard-coded 6. The trailing `&` continues a statement onto the next line. Each improvement removes a category of bugs: misspelled variables, wrong loop bounds, and silent loss of precision.

Feature	Legacy habit	Modern habit
Source form	Fixed columns, `.f` files	Free form, `.f90` files
Declarations	Implicit typing by first letter	`implicit none` everywhere
Precision	`REAL*8`, `DOUBLE PRECISION`	`real(dp)` with `dp = real64`
Loops	Labeled `DO 10 … 10 CONTINUE`	`do … end do` or array syntax
Shared data	`COMMON` blocks	Modules with `use`
Control flow	`GOTO` and arithmetic `IF`	`if`, `select case`, `exit`, `cycle`

TRY IT

Look at the two rainfall programs and list every line in the legacy version that has no counterpart in the modern version. Then list every safety feature the modern version adds. You should find at least three of each. When you finish Chapter 2 you can compile both and confirm they print the same numbers.

Fortran Compared with Python, C++, and Julia

No language is best at everything, and choosing Fortran for the wrong job wastes time. Here is an honest comparison with the three languages scientists most often weigh against it. We avoid quoting benchmark numbers on purpose: speed depends heavily on the problem, the compiler, the libraries, and the programmer, and published comparisons rarely transfer to your own code. The useful question is what each language makes easy.

Python

Python is excellent for exploration, plotting, data cleaning, scripting workflows, and gluing tools together. Libraries such as NumPy and SciPy give it fast array operations because their heavy lifting is done in compiled code, some of which is Fortran. Python becomes awkward when your algorithm needs explicit loops over millions of elements, when each step depends on the previous one (as in time-stepping simulations), or when you need predictable performance without restructuring the algorithm to fit library calls. A common and very effective pattern is to write the numerical core in Fortran and drive it from Python. Chapter 19 shows how.

C++

C++ is a powerful general-purpose language with a huge ecosystem, strong support for generic programming, and wide use in industry. It can match Fortran's speed, but getting there often requires more expertise: careful handling of aliasing, choosing or building a multidimensional array library, and understanding a large and complex language. If your project is mostly numerical arrays and formulas, Fortran usually lets a scientist write fast code with less language machinery. If your project is a large application with complex data structures, user interfaces, or heavy use of system libraries, C++ is often the better home.

Julia

Julia is a younger language designed for technical computing. It combines interactive use with just-in-time compilation and has an expressive type system and a growing package ecosystem. It is a strong choice for new projects that value interactivity and rapid development. Fortran's advantages over Julia are mostly about stability and context: decades of existing code and libraries, a formal international standard with several independent compilers, ahead-of-time compiled executables with no warm-up phase, and long-standing support on supercomputers.

A practical decision guide

- **Choose Fortran** when the core of the work is array-heavy numerical computation, when you must maintain or extend existing Fortran code, when the code will run for a long time on clusters, or when you want a stable language that will compile in the same way for many years.
- **Choose Python** for analysis, visualization, workflow scripts, and prototypes, and call compiled code for the expensive parts.
- **Choose C++** for large applications that need complex data structures, system integration, or a big general-purpose ecosystem.
- **Choose Julia** for new interactive technical projects where its ecosystem already covers your needs.

Mixing languages is normal. Many research groups use Python for input preparation and plotting and Fortran for the solver. You do not need to pick a side for your whole career, only for each component.

> **WARNING**
>
> Be skeptical of claims that one language is "always" some number of times faster than another. Such numbers usually come from a specific benchmark on a specific machine. When performance matters, measure your own program. Chapter 25 shows how to time code with `system_clock` and `cpu_time`.

Common misconceptions

Fortran's long history has left behind a few beliefs that are no longer true. It is worth clearing them up before you start, because they shape how people approach the language.

- **"Fortran is a dead language."** The language has an active international standards committee, new revisions appear regularly, and several independent compilers are maintained, including the free gfortran and LLVM flang and Intel's ifx. An open community also maintains modern tools such as the Fortran Package Manager, which you will use in Chapter 2.
- **"Fortran means uppercase code full of GOTO."** That describes the style of many old programs, not the language itself. Modern Fortran has block-structured control flow, modules, derived types, and procedures that read much like any other structured language.
- **"Fortran cannot do anything except arithmetic."** Modern Fortran handles strings, files, dynamic memory, object-oriented design, and calls to and from C. It is not the best tool for writing a web server, but it is perfectly capable of building a complete scientific application with input parsing, error handling, and tests.
- **"You need old Fortran knowledge to learn modern Fortran."** You do not. This book starts from modern style and only shows the legacy forms when you need to read them.

The honest weakness of Fortran is its ecosystem outside numerical computing. There are fewer general-purpose libraries than for Python or C++, fewer online examples written in a modern style, and less help from tooling than in some mainstream languages. These gaps are shrinking, and they matter less when Fortran is used for what it does best: the computational core of a scientific program.

Meet the Riverbend Environmental Lab

Throughout this book we follow a single, realistic project. The Riverbend Environmental Lab is a fictional university research group that monitors the Alder River, also fictional, from its upland source to the sea. You have just joined the lab as a new research programmer. Your job is to turn the lab's growing pile of measurements and scripts into reliable, fast, well-tested Fortran code.

The people

- **Dr. Lena Okafor** is the lead hydrologist and principal investigator. She designs the field campaigns and the lab's river-flow model, and she wants results she can defend in a paper.
- **Tomás Reyes** is the lab's research software engineer. He cares about build systems, testing, and code that other people can maintain. He will often review your code.
- **Mei Chen** is a PhD student who collects field data and analyzes storm events. She is the first user of most of the tools you write.

The stations

The lab operates four monitoring stations along the river:

ID	Name	Position	Elevation (m)
RB01	Upper Falls	Upstream	412
RB02	Mill Bridge	Middle reach	268

RB03	Cedar Bend	Lower reach	151
RB04	Harbor Mouth	Estuary	3

Each station records hourly readings of rainfall, water level, flow, and water temperature. The readings arrive in a file called `data/readings.csv` with this header line and rows like the one below:

```
station_id,timestamp,rain_mm,level_m,flow_m3s,temp_c
RB02,2026-04-03T06:00,1.8,1.42,23.6,9.4
```

A missing measurement is written as `-999.0`. Our programs will need to recognize that value and leave it out of statistics, which is a classic source of bugs when forgotten.

The first data set

In the early chapters we work with a small sample: six hours of readings from Mill Bridge (RB02) on 3 April 2026, from 00:00 to 05:00.

Hour	rain_mm	level_m	temp_c
00:00	0.0	1.10	8.9
01:00	0.4	1.11	8.8
02:00	2.6	1.15	8.8
03:00	5.2	1.24	8.7
04:00	3.1	1.33	8.9
05:00	0.8	1.38	9.1

The total rainfall is 12.1 mm, with a peak of 5.2 mm in the hour starting at 03:00. You can already see a storm moving through: the rain builds and fades, while the river level keeps rising because water takes time to travel through the catchment. That delay is exactly what the lab's model captures.

Where the project is heading

Dr. Okafor's main scientific tool is a linear-reservoir rainfall–runoff model for the Mill Bridge catchment. It treats the catchment as a single store of water that fills with rain and drains into the river. In Chapter 23 we solve the model's differential equation numerically, and in Chapter 28 we assemble everything into a complete, tested river-flow model with modules, derived types, file input and output, and parallel loops. Like many labs, Riverbend also has a few older programs written in fixed-form FORTRAN 77 that nobody wants to touch; in Chapter 27 you will learn how to bring that kind of code up to date safely.

Along the way, the code is organized into modules with consistent names: `riverbend_kinds` for precision, `riverbend_stats` for statistics, `riverbend_io` for file handling, `riverbend_station` for the station and reading types, `riverbend_model` for the reservoir model, and `riverbend_numerics` for numerical methods. You will build each one in the chapter where its ideas appear.

How to Use This Book

The book has six parts. Part I gets you installed and writing small programs with correct types and output. Part II covers arrays, strings, and floating-point arithmetic, the building blocks of numerical work. Part III teaches you to structure programs with procedures, modules, derived types, and files. Part IV covers modern features: object-oriented design, generic programming, error handling, interoperability with C and Python, and testing. Part V applies everything to numerical methods, and Part VI covers parallel computing, legacy code, and the final case study.

If you are new to Fortran, read Parts I to III in order; each chapter builds on the previous one. If you already know older Fortran, skim Part I, read Chapter 3 and Chapter 10 carefully because precision habits matter, and then jump to the parts you need. If you maintain legacy code, Chapter 27 can be read early, but it assumes you know modules and procedures from Part III.

Conventions

- Code listings that are complete files have a caption above them showing the file name, such as `src/riverbend_stats.f90`.
- When we show exactly what a program prints, the program uses an explicit format so the output is the same on every compiler. When we use the quick list-directed form `print *`, we remind you that the spacing varies by compiler.
- Every program uses `implicit none`, and from Chapter 3 onward all real numbers use the kind `dp` from the `riverbend_kinds` module.
- Boxes marked **Tip**, **Note**, and **Warning** highlight advice, background, and common mistakes. **Try It** boxes suggest small experiments; do them, because typing and breaking code teaches more than reading it.

Work the exercises

Every chapter ends with practice exercises labeled Easy, Medium, or Challenge, followed by worked solutions for a selection of them. Try each exercise before reading the solution. The Medium and Challenge exercises often extend the Riverbend code, so solving them gives you a growing toolkit that the later chapters build on.

TIP

Keep a single working folder for the book, with one subfolder per chapter. From Chapter 2 onward you will also keep an fpm project called `riverbend` that grows chapter by chapter. Put it under version control with Git from the first day; being able to see what changed is the cheapest debugging tool there is.

TRY IT

Before moving on, write down one numerical task from your own field that you would like to program in Fortran by the end of the book. As you read, note which chapter covers each piece you need. That list will keep your reading focused.

Practice Exercises

Some exercises ask you to write programs. If you have not installed a compiler yet, sketch the code on paper now and run it after Chapter 2.

1. **[Easy]** In one or two sentences each, explain three language features that help Fortran compilers produce fast numerical code.
2. **[Easy]** Using the station table, compute the elevation drop between each pair of neighboring stations (RB01 to RB02, RB02 to RB03, RB03 to RB04) and the total drop from Upper Falls to Harbor Mouth.
3. **[Easy]** Match each feature to the standard that introduced it: modules, coarrays, type-bound procedures, free-form source, `do concurrent`, C interoperability.
4. **[Easy]** For each of the following tasks, say whether you would start in Fortran, Python, C++, or Julia, and give one reason: (a) plotting a month of Riverbend data; (b) a 2-D diffusion solver that runs for hours; (c) extending a FORTRAN 77 groundwater model; (d) a desktop application with a complex user interface.
5. **[Medium]** In the legacy program `rain77.f`, suppose a programmer mistypes line 8 as `TOTAL = TOTL + R(I)`. Explain what happens when the program compiles and runs, and why the modern version would catch the mistake.
6. **[Medium]** Using the sample data, compute by hand the change in river level from 00:00 to 05:00 and the mean water temperature over the six hours. Keep three decimal places.
7. **[Medium]** Modify the modern program `rain_summary.f90` so that it also prints the largest hourly rainfall, using the intrinsic function `maxval`. Write the new `print` statement with an explicit format.
8. **[Medium]** Rewrite the following fixed-form loop in modern free form with a `do … end do` loop: `DO 20 I = 1, 6`, then `S = S + R(I)**2`, then `20 CONTINUE`. Then replace the whole loop by one assignment using array syntax.

9. **[Challenge]** Write a modern program that stores the four station elevations in an array and prints, using array sections like `level_rise.f90`, the elevation drop between neighboring stations. Show the exact output for your format.
10. **[Challenge]** Mei suggests rewriting the whole lab toolset in Python because "Python is easier." Tomás suggests keeping the model in Fortran. Write a short, balanced paragraph (no invented statistics) recommending a division of work between the two languages for the Riverbend project.

Selected Solutions

Exercise 2

The drops are 412 − 268 = 144 m, 268 − 151 = 117 m, and 151 − 3 = 148 m. The total from Upper Falls to Harbor Mouth is 412 − 3 = 409 m, which also equals 144 + 117 + 148. Checking that the pieces add up to the whole is a habit worth building for every calculation you program.

Exercise 5

Without `IMPLICIT NONE`, FORTRAN 77 gives any undeclared name a type based on its first letter: names starting with I to N are integers, everything else is real. So `TOTL` becomes a new real variable that was never assigned a value. Its starting value is undefined; on many systems it happens to be zero, in which case each pass through the loop sets `TOTAL` to just `R(I)`, and the program prints 0.800 (the last reading) as the total with no error at all. In the modern version, `implicit none` makes the compiler reject `totl` as an undeclared name, so the mistake never reaches a run.

Exercise 7

Add one statement after the existing `print`:

```
   print '(a, f8.3)', 'Peak (mm)  =', maxval(rain_mm)
```

The new line prints `Peak (mm) = 5.200`. The function `maxval` returns the largest element of an array; its partner `maxloc`, which returns the position, appears in Chapter 7.

Exercise 9

```
program elevation_drops
   implicit none
   integer :: elev_m(4), drop_m(3)

   elev_m = [412, 268, 151, 3]
   drop_m = elev_m(1:3) - elev_m(2:4)
   print '(a, 3i5)', 'Drops (m):', drop_m
   print '(a, i5)', 'Total (m):', sum(drop_m)
end program elevation_drops
```

```
Drops (m):  144  117  148
Total (m):  409
```

Elevations are whole meters, so an integer array is fine here. The section `elev_m(1:3)` holds the upstream station of each pair and `elev_m(2:4)` the downstream one, so their difference is the drop. The edit descriptor `3i5` prints three integers, each in a field five characters wide; Chapter 4 explains formats in detail.

Exercise 10

A reasonable recommendation: keep the rainfall–runoff model, the diffusion simulations, and any loop-heavy statistics in Fortran, where they are fast, compiled ahead of time, and easy to run on the lab's cluster. Use Python for data cleaning, exploratory analysis, plotting, and workflow scripts that run the Fortran programs and collect their output. The boundary between them should be simple, either well-defined data files or a small interface of the kind described in Chapter 19. This uses each language where it is strongest and avoids rewriting validated code without a reason.

Key Takeaways

- Fortran was designed for numerical computation and still excels at array math, mathematical formulas, and precise control of precision.
- Its speed comes from concrete properties: arrays known to the compiler, strict aliasing rules, and a small, predictable language.
- The language evolved through FORTRAN 77, Fortran 90/95, Fortran 2003, 2008, 2018, and 2023; Fortran 90 was the biggest leap.
- Modern style means free form, `implicit none`, explicit kinds, modules, and array syntax instead of fixed columns, implicit typing, and `COMMON`.
- Choose languages per component: Fortran for numerical cores, Python for analysis and glue, C++ for large general applications, Julia for interactive technical projects.
- The Riverbend Environmental Lab, with its four stations on the Alder River, is the running example for the entire book.
- Do the exercises: typing, compiling, and breaking code is how the ideas stick.