Dragos B. Chirila · Gerrit Lohmann

Introduction to Modern Fortran for the Earth System Sciences

Springer

Introduction to Modern Fortran for the Earth System Sciences

Dragos B. Chirila · Gerrit Lohmann

Introduction to Modern Fortran for the Earth System Sciences

Springer

Dragos B. Chirila
Gerrit Lohmann
Climate Sciences, Paleo-climate Dynamics
Alfred-Wegener-Institute
Bremerhaven
Germany

ISBN 978-3-642-37008-3 ISBN 978-3-642-37009-0 (eBook)
DOI 10.1007/978-3-642-37009-0

Library of Congress Control Number: 2014953236

Springer Heidelberg New York Dordrecht London

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

We dedicate this text to the contributors (too numerous to acknowledge individually) of the free- and open-source-software community, who created the tools that enabled our work.

Preface

"*Consistently separating words by spaces became a general custom about the tenth century A.D., and lasted until about 1957, when FORTRAN 77 abandoned the practice.*"

(Fortran 77 4.0 Reference Manual, Sun Microsystems, Inc)

Since the beginning of the computing age, researchers have been developing numerical Earth system models. Such tools, which are now used for the study of climate dynamics on decadal- to multi-millennial timescales, provide a virtual laboratory for the numerical simulation of past, present, and future climate transitions and ecosystems. In a way, the models bridge the gap between theoretical science (where simplifications are necessary to make the equations tractable) and the experimental science (where the full complexity of nature manifests itself, as multiple phenomena often interact in nonlinear ways, to form the final signal measured by the apparatus). Models provide intermediate subdivisions between these two extremes, allowing the scientist to choose a level of detail that (ideally) strikes a balance between accuracy and computational effort.

The development of models has accelerated in the last 50 years, largely due to decreasing costs of computing hardware and emergence of programming languages accessible to the non-specialist. Fortran, in particular, was the first such language targeting scientists and engineers, therefore it is not surprising that many models were written using this technology. To many, however, this long history also causes Fortran to be associated with the punched cards of yesteryear and obsolete software practices (hence the quotation above). A programming language, however, evolves to meet the demands of its community, and such was also the case with Fortran: object-oriented and generic programming, a rich array language, standardized interoperability with the `C`-language, free-format (!), and many more features are now available to Fortran programmers who are willing to take notice.

Unfortunately, many of the newer features and software engineering practices that we consider important are only discussed in advanced books or in specialized reference documentation. We believe this unnecessarily limits (or delays) the exposure of beginning scientific programmers to tools, which were ultimately designed to make their work more manageable. This observation motivated us to

write the present book, which provides a short "getting started" guide to modern Fortran, hopefully useful to newcomers to the field of numerical computing within *Earth system science* (ESS) (although we believe that the discussion and code examples can also be followed by practitioners from other fields). At the same time, we hope that readers familiar with other programming languages (or with earlier revisions of the Fortran-standard) will find here useful answers for the "How do I do X in modern Fortran?" types of questions.

Chapters Outline

In Chap. 1, we start with a brief history of Fortran, and succinctly describe the basic tools necessary for working with this book. In Chap. 2, we expose the fundamental elements of programming in Fortran (variables, I/O, flow-control constructs, the Fortran array language, and some useful intrinsic procedures). In Chap. 3, we discuss the two main approaches supported by modern Fortran for structuring code: *structured programming* (SP) and *object-oriented programming* (OOP). The latter in particular is a relative newcomer in the Fortran world.

The example-programs (of which there are many in the book) accompanying the first three chapters are intentionally simple (but hopefully still not completely uninteresting), to avoid obfuscating the basic language elements. After practicing with these, the reader should be well equipped to follow Chap. 4, where we illustrate how the techniques from the previous chapters may be used for writing more complex applications. Although restricted to elementary numerical methods, the case studies therein should resemble more of what can be encountered in actual ESS models.

Finally, in Chap. 5 we present additional techniques, which are especially relevant in ESS. Some of these (e.g., namelists, interoperability with `C`, interacting with the *operating system* (OS)) are Fortran features. Other topics (I/O with *NETwork Common Data Format* (`netCDF`), shared-memory parallelization, build systems, etc.) are outside the scope of the Fortran language-standard, but nonetheless essential to any Fortran programmer (the `netCDF` is ESS-specific).

Language-Standards Covered

The core of the book is based on Fortran 95.[1] Building upon this basis, we also introduce many newer additions (from Fortran 2003 and Fortran 2008[2]), which complete the discussion or are simply "too good to miss"—for example OOP,

[1] This was, at the time of writing, the most recent version with ubiquitous compiler support.

[2] Many compilers nowadays have complete or nearly complete support for these newer language-standard revisions.

interoperability with the C-language, OS integration, newer refinements to the Fortran array language, etc.

Disclaimers

- Given the wide range of topics covered and the aim to keep our text brief, it is obvious that we cannot claim to be comprehensive. Indeed, good monographs exist for many topics, which we only superficially mention (many further references are cited in this text).
- Finally, we often provide advice related to what we consider good software practices. This selection is, of course, subjective, and influenced by our background and experiences. Specific project conventions may require the reader to adapt/ignore some of our recommendations.

How to Use this Book

Being primarily a compact guide to modern Fortran for beginners, this book is intended to be read from start to finish. However, one cannot learn to program effectively in a new language just by reading a text—as in any other "craft", practice is the best way to improve. In programming, this implies reading and writing/testing as much code as possible. We hope the reader will start applying this philosophy while reading this book, by typing, compiling, and extending the code samples provided.[3]

Readers with programming experience may also use "random access," to select the topics that interest them most—the chapters are largely independent, with the exception of Chap. 5, where several techniques are demonstrated by extending examples from Chap. 4.

Due to the "breadth" of the book, many technical aspects are covered only superficially. To keep the main text brief, we opted to provide as footnotes suggestions for further exploration. Unfortunately, this led to a significant number of footnotes at times; the reader is encouraged to ignore these, at least during a first reading, if they prove to be a distraction.

[3] Nonetheless, the programs are also available for download from SpringerLink. The authors also provide a code repository on GitHub: assuming a working installation of the git version-control system is available, the code repository can be "cloned" with the command:

```
git clone https://github.com/dchirila/imf_ess.git
```

Acknowledgments

The idea of writing this book crystallized in the spring of 2012. Almost 2 years later, we have the final manuscript in front of us. Contributions from many people were essential during this period. They all helped in various ways, through discussions about the book and related topics, requests for clarifications, ideas for topics to include, and corrections of our English and of other mistakes, greatly improving the end result. In particular, we acknowledge the help of many (past and present) colleagues from the *Climate Sciences* division at *Alfred-Wegener-Institut, Helmholtz-Zentrum für Polar- und Meeresforschung* (AWI)—especially Manfred Mudelsee, Malte Thoma, Tilman Hesse, Veronika Emetc, Sebastian Hinck, Christian Stepanek, Dirk Barbi, Mathias van Caspel, Sergey Danilov, and Dmitry Sidorenko. We thank Stefanie Klebe for a very thorough reading of the final draft, which significantly improved the quality of the book.

In addition to our AWI colleagues, we received valuable feedback from Li-Shi Luo, Miguel A. Bermejo, and Dag Lohmann.

Our editors from Springer were very helpful during the writing of this book. In particular, we thank Marion Schneider, Johanna Schwarz, Carlo Schneider, Marcus Arul Johny, Ashok Arumairaj, Janet Sterritt, Agata Oelschlaeger, Dhanusha M. and Janani J. for kindly answering our questions and for their support.

Finally, we would like to thank our families and friends, who contributed with encouragement, support, and patience while we worked on this project.

Bremerhaven, Germany, May 2014

Dragos B. Chirila
Gerrit Lohmann

Contents

Acronyms

ADE	Alternating-direction explicit
ADT	Abstract data type
API	Application programming interface
BC	Boundary condition
CA	Cellular automata
CAF	Co-array Fortran (http://www.co-array.org/)
CF	Climate forecast (http://cf-pcmdi.llnl.gov/documents/cf-conventions)
CFL	Courant-Friedrichs-Levy
CLI	Command line interface
CPU	Central processing unit
DAG	Directed acyclic graph
DSL	Domain-specific language
DT	Derived Data Type
EBM	Energy balance model
EBNF	Extended Backus-Naur form
ESS	Earth system science
FD	Finite differences
FE	Finite elements
FV	Finite volumes
GP	Generic programming
GPGPU	General-purpose graphics processing unit
GUI	Graphical user interface
HDD	Hard disk drive
HPC	High performance computing
HLL	High-level language
I/O	Input/output
IC	Initial condition
ID	Identifier
IDE	Integrated development environment

ILP	Instruction-level parallelism
LBM	Lattice Boltzmann method
LGCA	Lattice gas cellular automata
LHS	Left-hand side
MPI	Message Passing Interface
MRT	Multiple relaxation times
NAS	Network-attached storage
ODE	Ordinary differential equation
OOP	Object-oriented programming
OpenCL	Open Computing Language
OpenMP	Open MultiProcessing
OS	Operating system
PDE	Partial differential equation
PDF	Particle distribution function
PGAS	Partitioned Global Address Space (http://www.pgas.org/)
RAM	Random access memory
RB	Rayleigh-Bénard
RHS	Right-hand side
RNG	Random number generator
RPM	Revolutions per minute
SIMD	Single instruction, multiple data
SP	Structured programming
SPMD	Single program, multiple data
TDD	Test-driven development
TRT	Two relaxation times

Fortran Compilers

`gfortran`	GNU Fortran Compiler (see entry on `gcc`)
`ifort`	Intel Fortran Compiler® (http://software.intel.com/en-us/fortran-compilers)

Profiling Tools

`gprof`	GNU Profiler (part of `binutils`) (http://www.gnu.org/software/binutils/)
`VTune`	Intel VTune Amplifier XE 2013 (https://software.intel.com/en-us/intel-vtune-amplifier-xe)

Other Software Utilities

`bash`	Bourne-again shell (http://www.gnu.org/software/bash)
`CMake`	Cross Platform Make (http://www.cmake.org)

Cygwin	http://www.cygwin.com/index.html
gcc	GNU Compiler Collection (http://gcc.gnu.org)
ld	GNU linker (http://www.gnu.org/software/binutils)
gmake	GNU Make (http://www.gnu.org/software/make)
MinGW	Minimalist GNU for Windows (http://www.mingw.org)
SCons	Software Construction tool (http://www.scons.org)

Visualization/Post-processing Tools

CDO	Climate Data Operators (https://code.zmaw.de/projects/cdo)
GMT	Generic Mapping Tools (http://gmt.soest.hawaii.edu)
gnuplot	http://www.gnuplot.info
NCO	netCDF Operators (http://nco.sourceforge.net)
ParaView	Parallel Visualization Application (http://www.paraview.org)

Operating Systems

AIX	IBM Advanced Interactive eXecutive
Linux	GNU/Linux
OSX	Mac OS X®
Windows	Microsoft Windows®
Unix	Unix® (http://www.unix.org)

Text Editors

Emacs	GNU Emacs text editor (http://www.gnu.org/software/emacs)
gedit	Gedit text editor (http://projects.gnome.org/gedit)
joe	Joe's Own Editor (http://joe-editor.sourceforge.net)
Kate	Kate text editor (http://kate-editor.org)
Vim	Vim text editor (http://www.vim.org)

Software Libraries

ACML	Core Math Library (http://developer.amd.com/tools/cpu-development/amd-core-math-library-acml)
ATLAS	Automatically Tuned Linear Algebra Software (http://math-atlas.sourceforge.net)
BLAS	Basic Linear Algebra Subprograms
Boost.Program_Options	http://www.boost.org/libs/program_options
ESSL	Engineering Scientific Subroutine Library

fruit	FORTRAN Unit Test Framework (http://sourceforge.net/projects/fortranxunit)
GAMS	Guide to Available Mathematical Software (http://gams.nist.gov)
HDF5	Hierarchical Data Format—Version 5 (http://www.hdfgroup.org/HDF5/)
JAPI	Java Application Programming Interface (http://www.japi.de)
LAPACK	Linear Algebra PACKage (http://www.netlib.org/lapack)
MKL	Intel® Math Kernel Library (http://software.intel.com/en-us/intel-mkl)
netCDF	NETwork Common Data Format (http://www.unidata.ucar.edu)
netlib	http://www.netlib.org
Winteracter	http://www.winteracter.com
Zenity	https://help.gnome.org/users/zenity/stable

Other Programming Languages

C	http://en.wikipedia.org/wiki/C_(programming_language)
C++	http://en.wikipedia.org/wiki/C%2B%2B
COBOL	Common Business Oriented Language
Java	http://www.java.com/en/
MATLAB	Matrix Laboratory® (http://www.mathworks.com)
octave	GNU Octave (http://www.gnu.org/software/octave)
Pascal	http://en.wikipedia.org/wiki/Pascal_(programming_language)
Python	http://www.python.org
R	The R Project for Statistical Computing (http://r-project.org)

Version Control Software

git	http://www.git-scm.com
mercurial	http://mercurial.selenic.com
monotone	http://www.monotone.ca
subversion	http://subversion.apache.org

Earth System Science Models

Planet Simulator	http://www.mi.uni-hamburg.de/Planet-Simul.216.0.html?&L=3

Organizations and Companies

AMD	Advanced Micro Devices Inc.
ANL	Argonne National Laboratory (http://www.anl.gov)
Apple	Apple Inc.
ASCII	American Standard Code for Information Interchange
AWI	Alfred-Wegener-Institut, Helmholtz-Zentrum für Polar- und Meeresforschung (http://www.awi.de)
GNU	GNU project—software project backed by the Free Software Foundation (FSF); the (recursive) acronym stands for GNU's Not Unix! (http://www.gnu.org)
IBM	International Business Machines Inc.
Intel	INTegrated ELectronics Inc.
OGC	Open Geospatial Consortium (http://www.opengeospatial.org/standards/netcdf)
UCAR	University Corporation for Atmospheric Research (https://www2.ucar.edu)
WMO	World Meteorological Organization (http://www.wmo.int)

Conventions in this Text

The following conventions are used for the code samples:

1. *Formatting and color scheme*
   Programs and code samples that would normally be typed in an editor, are shown in boxes, with the following conventions in place:

- *keywords*[4]: dark gray, bold font
- *character strings*: medium gray, normal font
- *comments*: medium gray, italic font

2. *Code placeholders*

- *Optional items* are emphasized using square brackets. When the reader wishes to include them within programs, the brackets should be removed.

[4] We choose to typeset Fortran keywords with lowercase letters, although the language is case-insensitive everywhere except inside character strings (so `PROGRAM`, `program` or `PrOgRaM` is all the same to the compiler).

- *Mandatory items* that should be supplied by readers, as well as *invisible characters* are emphasized using angle brackets, as in:

```
if( <logical expression> ) then
    print*, ''Expression was .true.''
end if

<Enter>
<Space>
```

  It should be easy to infer from the context what these angle bracket expressions should be replaced with.
- *Combinations* of *optional* and *mandatory* items are sometimes highlighted by nesting of square and angle brackets, to distinguish the fact that including some items may unlock additional possible combinations.

3. With the exception of small snippets, *code listings* are accompanied by a caption, indicating the corresponding file in the source code tree available for download. Line numbers are only shown when they are specifically referenced in the text.
4. Where interaction with the Operating System (OS) is illustrated, we describe the process for the *GNU/Linux* (`Linux`) platform, using *Bourne-again shell* (`bash`), since this environment is easily accessible. Commands corresponding to such tasks are marked by a leading `$`-character (default shell-prompt in `Linux`); only the part after this marker should be typed.
5. Exercises are typeset on a dark-gray background, to distinguish them from the rest of the text.
6. Several notes appear as framed boxes on a light gray background.
7. *Naming conventions* It is usually considered good practice to adopt some rules for naming entities that are part of the program code. Although different developers may prefer a different set of such rules, it is generally a good idea to use a single convention *consistently* within a project, to reduce the effort required for understanding the code. Our particular conventions are explained below.

Naming Rules for Data

- Variables (both scalars or arrays) are named as things (nouns) or attributes (adjectives). When they consist of multiple words, camel-case is used, starting with a lowercase letter:

```
temperature, numIterations
```

- Variables that are part of a user-defined type follow the same rules as above, except that the first letter is always a lowercase “m”:

```
mNx, mOutFilePrefix
```

- Constants are written in uppercase, and when they are composed of multiple words they are separated by underscores:

```
PI, MAX_NUM_ITERATIONS
```

- User-defined types (analogs of C++ classes) are named as variables (camel-case nouns), except that they begin with a capital letter:

```
Vec2D, OceanBox
```

Naming Rules for Procedures (Functions, Subroutines)

- Normal procedures (i.e., those which are not bound to a specific user-defined type) look similar to usual variables, *except* that they contain verbs, to emphasize the function of the procedure:

```
swap, isPrime, computeAverageTemperature
```

- Procedures that are bound to a specific type follow the rules above, but also have the name of the type at the end:

```
swapReal, getMagnitudeVec2D
```

This rule is introduced mostly to avoid naming collisions, when the same type-bound procedure name makes sense for several types (but their implementation differs). For simplifying the calling of these procedures, we usually define shorter aliases (which omit the type-name), as explained in Chap. 3.

Naming Rules for Modules and Source Code Files

- For naming Fortran modules which *do not* encapsulate a user-defined type, we use nouns and camel-case (first letter being uppercase):

```
Utilities, NumericKinds
```

A common guideline is to place each module in a separate file; for example, the modules above would be placed in files `Utilities.f90` and `NumericKinds.f90`, respectively. However, we do not adhere to this rule until later in the book, after explaining how to work with projects composed of multiple files, in Sect. 5.1.

- Fortran modules that also encapsulate a user-defined type are named after the type, with the added prefix `_class`:

```
Vec2D_class, OceanBox_class
```

  When these are placed in distinct files, the filename is composed of the module name, with the added extension `.f90`. For example, the modules above would be placed in files `Vec2D_class.f90` and `OceanBox_class.f90`.

Chapter 1
General Concepts

This chapter introduces the Fortran programming language in the context of numerical modeling, and in relation to other languages that the reader may have experience with. Also, we discuss some technical requirements for making the best use of this book, and provide a brief overview of the typical workflow for writing programs in Fortran.

1.1 History and Evolution of the Language

In the 1950s, a team from *International Business Machines Inc*. (`IBM`) labs led by John Backus created the Fortran ("mathematical FORmula TRANslation system")[1] language. This was the first *high-level language* (HLL) to become popular, especially in the domain of numerical and scientific computing, for which it was primarily designed. Prior to this development, most computer systems were programmed in assembly languages, which only add a thin wrapper on top of raw machine language (generally leading to software which is not portable and more difficult to maintain). Fortran was widely adopted due to its increased level of abstraction, which made Fortran programs orders of magnitude more compact than corresponding assembly programs. This popularity, combined with intentional simplifications of the language (for example, lack of pointer type in earlier versions), encouraged the development of excellent optimizing compilers, making Fortran the language of choice for many demanding scientific applications.

This is also the case for *Earth system science* (ESS), where Fortran is to date the most used programming language. The reasons are simple: there is a huge body of tested Fortran routines, and the language is very suitable for coding physical

[1] The reader may sometimes encounter the name of the language spelled in all capitalized (as in `FORTRAN`), usually referring to the early versions of the language, which officially supported only uppercase letters to be used in programs. This shortcoming was corrected by the later revisions, with which we are concerned in the present text.

D.B. Chirila and G. Lohmann, *Introduction to Modern Fortran for the Earth System Sciences*, DOI 10.1007/978-3-642-37009-0_1

equations. Early model implementations based on Fortran started in the mid of last century (see e.g. Bryan [1], Platzman [4], Lynch [3] and references therein). The models predicted how changes in the natural factors that control climate, such as ocean and atmospheric currents and temperature, could lead to climate change. Climate models are intended to provide a user-friendly and powerful framework for simulating real or idealized flows over wide-ranging scales and boundary conditions. With its good support for modular programming, Fortran proved to be well suited for these tasks.

Certainly, many other languages were introduced over the last 60 years (such as the `COBOL`, `Pascal`, `C`, `C++`, `Java`, etc.), some offering innovative facilities for expressing algorithm abstractions (such as object-oriented or generic programming). Interestingly, these languages did not supersede Fortran (at least not in the ESS community); instead, they inspired the Fortran language-standardization committee to incorporate such facilities through incremental revisions (Fortran 90, Fortran 95, Fortran 2003, and Fortran 2008 at the time of writing).

1.2 Essential Toolkit (Compilers)

Fortran is a compiled language, so an ASCII text editor and a compiler should be enough to get started. A popular compiler is the *GNU Fortran Compiler* (`gfortran`), which is freely available as part of the *GNU Compiler Collection* (`gcc`). For users of `Unix`-like systems, this should be easily available, either in the system's package manager (*GNU/Linux* (`Linux`)), or bundled within the XCode developer package for *Mac OS X*® (`OSX`). It is also possible to install `gfortran` on *Microsoft Windows*® (`Windows`) systems, using the *Minimalist GNU for Windows* (`MinGW`) or `Cygwin` systems.

Many other compilers exist, some offering useful features like more powerful code optimizers,[2] convenient debugging/profiling tools, and/or a user-friendly *integrated development environment* (`IDE`). It is not possible to cover the whole landscape here—please consult a local expert or system administrator for advice on a suitable compiler.

The example programs were tested with recent versions of `gfortran` and of the *Intel Fortran Compiler*® (`ifort`), on `Linux`. However, the programs should be easy to adapt to other recent compilers and/or platforms.

[2] For most supercomputers, the compilers are usually provided by the hardware vendor, which allows better tuning of the code to the features of the underlying machine.

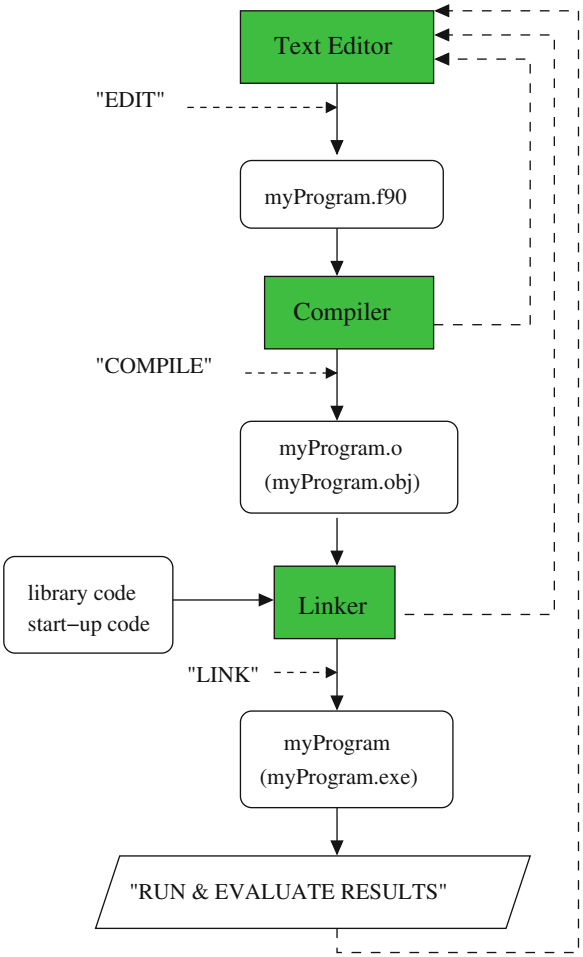

Fig. 1.1 Schematic of programming workflow in Fortran. Files are represented as *white rounded boxes*, and external programs as *green boxes*

1.3 Basic Programming Workflow

From a low-level perspective (i.e. leaving more abstract issues such as program design aside), development of Fortran programs[3] is represented schematically in Fig. 1.1.

[3] The terms "program" and "(source) code" are used interchangeably within this book; however, strictly speaking, "code" can also refer to program sub-modules, such as functions, while "program" usually refers to a complete application, which yields an executable file when processed by a compiler.

In the figure, the utilities are shown as green boxes. The process starts with a text editor,[4] where the user enters the program code.[5] Then, the compiler is invoked, passing the created file as an argument. In `Linux`, using `gfortran`, this would be achieved by typing the following command in a terminal window:

```
$ gfortran -c myProgram.f90
```

At this point, an additional file (`myProgram.o`) will be created. This contains machine code generated from `myProgram.f90` which does not contain, however, any code for libraries that may be needed by your program. It is the job of the linker to find the missing pieces and to produce the final, executable file. In `Linux`, the *GNU linker* (`ld`) is normally used for this purpose. For simplicity, it is better to perform the linking stage also through the compiler, which will call the linker with the appropriate options in the background:

```
$ gfortran -o myProgram myProgram.o
```

(in `Windows`, replace `myProgram.o` with `myProgram.obj`, and `myProgram` with `myProgram.exe`).

This step will create the executable program, which can be run with the command:

```
$ ./myProgram
```

The entire workflow seems deceivingly simple.[6] In reality, problems can appear at any stage (especially in nontrivial programs), which trigger the need to revise the program. These iterative improvements of the code are suggested by the dashed lines in Fig. 1.1. First, the compiler may refuse to produce object-code if the program does not follow the syntax of the language. Then, the linker may be unable to find the appropriate libraries to include. Finally, the program may crash, or it may run but produce unacceptable results. The beginner will usually encounter problems across all of these ranges. Fortunately, with some practice, the frequency of the (less interesting) compilation/linking errors decreases.

Compiling and linking in one step. So far, we separated the two phases for producing the program executable, to make the reader aware of the distinction (when

[4] Word processors are a poor choice here, since they focus on features like advanced formatting, which the compiler does not understand anyway; instead, a "bare bones" text editor, but with programming-related features like syntax highlighting and auto-completion, is recommended, for example: *gedit text editor* (`gedit`) or *Kate text editor* (`Kate`) are good starting points; *Vim text editor* (`Vim`), *GNU Emacs text editor* (`Emacs`) or *Joe's Own Editor* (`joe`) are more advanced choices, that may pay off on the longer term.

[5] Files containing modern Fortran source code usually have the extension `.f90`, but the reader may also encounter extensions `.f77`, `.f`, or `.for`, which correspond to older standards; likewise, some developers may use the extensions `.f95`, `.f03`, or `.f08`, to highlight use of features present in the latest revisions of the language—but this practice is discouraged by some authors (e.g. Lionel [2]). To avoid problems, filenames should also not contain whitespace.

[6] Indeed, this resembles the Feynman problem-solving algorithm: (a) write down the problem, (b) think very hard, and (c) write down the answer.

the executable fails to build properly, it is useful as a first step to determine if we face a compiler or linker error). However, for single-file programs, these steps can be combined in a single command:

```
$ gfortran -o myProgram myProgram.f90
```

For programs consisting of several files, compiling and linking by hand is impractical, and a build system becomes essential (discussed later, in Sect. 5.1).

References

1. Bryan, K.: A numerical method for the study of the circulation of the world ocean. J. Comput. Phys. **4**(3), 347–376 (1969)
2. Lionel, S.: Doctor Fortran in "Source Form Just Wants to be Free" (2013). http://software.intel.com/en-us/blogs/2013/01/11/doctor-fortran-in-source-form-just-wants-to-be-free
3. Lynch, P.: The origins of computer weather prediction and climate modeling. J. Comput. Phys. **227**(7), 3431–3444 (2008)
4. Platzman, G.W.: The ENIAC computations of 1950—gateway to numerical weather prediction. Bull. Am. Meteorol. Soc. **60**(4), 302–312 (1979)

Chapter 2
Fortran Basics

In this chapter, we introduce the basic elements of programming using Fortran. After briefly discussing the overall syntax of the language, we address fundamental issues like defining variables (of intrinsic type). Next we introduce *input/output* (I/O), which provides the primary mechanism for interacting with programs. Afterwards, we describe some of the flow-control constructs supported by modern Fortran (`if`, `case`, and `do`), which are fundamental to most algorithms. We continue with an introduction to the Fortran array-language, which is one of the strongest points of Fortran, of particular significance to scientists and engineers. Finally, the chapter closes with examples of some intrinsic-functions that are often used (for timing programs and generating pseudo-random sequences of numbers).

2.1 Program Layout

Every programming language imposes some precise syntax rules, and Fortran is no exception. These rules are formally grouped in what is denoted as a "context-free grammar",[1] which precisely defines what represents a valid program. This helps the compiler to unambiguously interpret the programmer's source code,[2] and to detect sections of source code which do not follow the rules of the language. For readability, we will illustrate some of these rules through code examples instead of the formal notation.

Below, we show the basic layout of a single-file Fortran program, with no procedures (these will be discussed later):

[1] For example, *extended Backus-Naur form* (EBNF).

[2] EBNF is also useful for defining consistent data formats and even simple *domain-specific languages* (DSLs).

D.B. Chirila and G. Lohmann, *Introduction to Modern Fortran for the Earth System Sciences*, DOI 10.1007/978-3-642-37009-0_2

```
program [program name]
  implicit none

  [ variable declarations [ initializations ] ]

  [ code for the program ]

end program [program name]
```

Any respectable language tutorial needs the classical "Hello World" example. Here is the Fortran equivalent:

```
program hello_world
  implicit none
  print*,"Hello, world of Modern Fortran!"
end program hello_world
```

Listing 2.1 `src/Chapter2/hello_world.f90`

This should be self-explanatory, except maybe for the `implicit none` entry, which instructs the compiler to ensure all used variables are of an explicitly defined type. It is strongly recommended to include this statement at the beginning of each program.[3] The same advice will apply to modules and procedures (discussed later).

Exercise 1 (*Testing your setup*) Use the instructions from Sect. 1.3 (adapting commands and compiler flags as necessary for your system) to edit, compile and execute the program above. Try separate compilation and linking first, then combine the two stages.

2.2 Keywords, Identifiers and Code Formatting

All Fortran programs consist of several types of *tokens*: *keywords* (reserved words of the language), *special characters*,[4] *identifiers* and *constant literals* (i.e. numbers, characters, or character strings). We will encounter some of the keywords soon, as we discuss basic program constructs. Identifiers are the names we assign to variables or constants. The first character of an identifier should be a letter (the rest can be

[3] This is related to a legacy feature, which could lead to insidious bugs. The take-home message for new programmers is to always use `implicit none`. The `-fimplicit-none` flag can be used, in principle, in `gfortran`, but this is also discouraged because it introduces an unnecessary dependency on compiler behavior.

[4] The special characters are (framed by boxes): `=`, `+`, `-`, `*`, `/`, `(`, `)`, `,`, `.`, `$`, `'`, `:`, ␣ (blank), `!`, `"`, `%`, `&`, `;`, `<`, `>`, `?`, `\`, `[`, `]`, `{`, `}`, `~`, `` ` ``, `^`, `|`, `#`, and `@`. Certain combinations of these are reserved for *operators* and *separators*.