# 1.

# Introduction

## 1.1 Acknowledgments

The authors thank the Numerical Algorithms Group Ltd. (NAG) who provided us with copies of their excellent compiler with which we could test our code. In particular, thanks go to Mr. Malcolm Cohen, Mr. Rob Holmes, Mr. Ian Hounam, Mr. Rob Meyer, Mr. Mike Modica, and Mr. John Morrissey.

The Portland Group provided us with a copy of their compiler. Special thanks go to Ms. Laura Gibon for arranging that.

We thank Mr. Art Lazanoff for the use of his network server system for our CVS repository.

We thank Mr. Dan Nagle who offered vigorous criticism and some good suggestions.

The following persons read over the manuscript; to them we owe our gratitude: Dr. Greg Brown, Dr. Charles Crawford, Mr. Ryan O'Kuinghttons, and Dr. James Hlavka.

Thanks go to Ms. Stacy L. Castillo at the IBM Corporate Archives for arranging permission to use the material for the frontispiece.

It was a great pleasure to work with our editors at Cambridge University Press: Ms. Heather Bergman, Ms. Lauren Cowles and Mr. David Jou.

## 1.2 Typographical Conventions

The following typographical conventions are used in this book:

- medium-weight serif font – normal text
  This sentence is written in the font used for normal text.
- ***bold italicized serif font*** – rules
- medium-weight sans serif font – computer code
- **bold sans serif font** – Fortran keywords
  Examples are the words "**null**," "**associated**," and "**save**" in Rule 74. Note that the font for keywords and for names from the computer code is used in the body of the normal text, not just in the code segment.
- *`medium-weight italicized serif font`* – terms from the either the Fortran 2003 or the Fortran 2008 Standard, References [39] and [43]

In source code listings, points of ellipsis (. . .) are used to indicate missing code – nonessential code that is left out for brevity and clarity.

## 1.3 Source Code Listings

For the most part the code examples in the book are short sections of code. As such, they cannot be compiled. In some instances, however, complete programs are presented as numbered listings. In both cases, comments explaining the key points are embedded in the code and then referred to in the text. These are marked and numbered, in both places, by "*Cmnt-*i*:," where *i* is the comment number. Here, for example, is one such comment and the following two lines of code, and then the explanation referring to it in the text:

```
  ! *Cmnt-1: Check arguments for sanity
select case (direction)
case ('forward', 'FORWARD', 'reverse', 'REVERSE')
```

Using Rule 63, the first argument check (*Cmnt-1) can be entirely eliminated . . .

# 2.

# General Principles

"The purpose of computing is not numbers. The purpose of computing is understanding."
– Hamming

**1.** ***Write programs that are clear to both the reader and the compiler.***

The first and foremost general principle of programming is clarity. From clarity comes the ability to test, to reuse, and to audit. One simple test is whether you believe you will be able to understand the code if you come back to it a year later.

Of course, you write programs to have the computer calculate something for you. And you know that the computer must be told exactly what to compute. Your program must completely and correctly specify what is to be computed.

If you are making a numerical calculation, you likely have some consideration for the efficiency of your calculation. The best way to gain efficiency is to first choose an efficient algorithm and then to write simple, clear, and logical code to implement it. Compared with more complex code, it is easier to understand and easier for the compiler to optimize.

In addition to writing code for the computer, you are also writing code for humans, yourself included. The purpose of the calculations and the methods used to do so must be clear.

To achieve these goals, write code that is as simple as possible. Use white space to aid your eye in following the calculation specified by the source code. Comment what cannot be understood from the code itself. The rules in this book follow from these ideas. Using them will promote consistency of visual layout, documentation, programming logic, and the naming of program entities. This consistency, in turn, increases clarity; the program is clear to a programmer, whether he or she is familiar with it or not, or experienced or not. Moreover, for the programmer who is charged with learning and modifying it, a program written in a consistent manner reduces the time required to get "up to speed."

Ways to document a program are described in Chapter 5. The term "self-documenting" is often used to describe code that conveys its design without excessive commentary. The names of both variables and named constants should indicate what they represent. The algorithms used in the program

should be familiar to anyone educated in the field for which the program has been written. We recommend the naming conventions in Chapter 4.

Clarity for the compiler is aided by simplicity and the use of language-defined structured constructs (e.g., rather than **go to** statements) to achieve program flow. Excessive branching, especially inside **do** loops, can often thwart optimization.

Avoid "cute tricks." They are often found in code that may conform to the standard, but take advantage of things like internal data representations, which are not standardized. These tricks have two problems: They obscure what you are trying to compute, from both the compiler and you, and they are very likely not portable, even among different compilers on the same hardware. Many uses of the intrinsic function **transfer** and bit manipulation intrinsic functions fall into this category.

A simple test for whether a program unit is clearly expressed is the Telephone Test (see Reference [45]). If the program can be read over the telephone, and completely understood by the listener, it is likely clear.

**2.** ***Write programs that can be efficiently tested.***

A program must produce correct results when provided valid input. Key to this is to emphasize error detection and correction as early as possible in the development and testing process. The earlier a problem is detected, the less it costs to fix it.

Several techniques can be used to help produce quality code with reduced debugging times. First, when writing code, take advantage of modern features that allow the compiler to detect errors at compile time. Two key items for Fortran programmers are the use of **implicit none**, for avoiding typographical mistakes, and the use of modules for packaging and, doing so, ensuring interface checking. The use of these two features is highly recommended for all code.

Second, it is desirable to modularize and code individual algorithms into procedures in a way that they can be independently tested and verified. Testing procedures independent of the entire application is called "unit testing," and the individual tests are called "unit tests." Unit tests are a fundamental tool for validating a procedure (see Reference [47]).

To be easily unit tested, a procedure should use a minimal number of variables outside its local scope. A test driver can then be written to present the target procedure with different combinations of arguments, and compare the actual returned results with known good results.

Each test driver should indicate in some manner, for example in a log file, each of the tests it has run, and a PASS or FAIL flag. Simple scripts can then be written to run each of the unit test drivers, and to summarize the results.

Regressions, code that worked previously but, after some changes have been made, no longer does, can be quickly spotted and repaired.

A third aspect of reliability is the rejection of inputs that are invalid. For example, it is often possible to include non-time-consuming tests of the input arguments for validity, such as ensuring that arrays have compatible sizes. The routine can then return an error code back to the caller instead of producing incorrect results. This is discussed in more detail in Section 6.4. By returning an error code, instead of aborting, unit tests may be written to test for bad inputs, as well as good inputs.

Finally, once individual components of an application have been tested, tests on complete problems can be made to verify the application as a whole. Inputs representing typical end-user problems can be provided that produce known good results. Tests should encompass the full range of allowable inputs. Tests of numerical algorithms should include very large and small values; decision algorithms should be tested with as many combinations, such as true or false conditions, as practical. For numerical algorithms that will break down with very large arguments, such as computing $\cos(10^{10})$, the documentation should specify to what degree it has been tested and the results of the tests. As an application is developed and maintained, the test base provides a valuable feedback mechanism to look for regressions.

**3.** ***Write programs that will scale to different problem sizes.***

Scalability is the property of a program to accommodate a wide range of problem sizes. In other words, a program should be able to handle small test cases using minimal computer resources, and, ideally, it should also be able to handle the largest problems that a given machine is capable of processing without changing any of the source code. Arrays and other data structures should be able to adjust themselves to any reasonable problem size. This can also result in greater efficiency through better use of cache memory.

Since Fortran 90, Fortran has supported various techniques – pointers, and allocatable variables, assumed shape and automatic arrays – for dynamically allocating memory based on problem size. These should be used instead of fixed dimensions wherever an array size may vary (instead of the earlier practice of hard-coding maximum sizes).

Scalability is also often used to describe how well a program takes advantage of multiple processors and the multiple cores of modern processors. Many schemes for executing portions of a code in parallel have been implemented by the Fortran community, including OpenMP and MPI. Section 12.2.2 covers OpenMP in more detail; Section 12.2.3 does the same for MPI. Fortran 2008 introduces the **do concurrent** variant of the **do** construct for shared memory parallelism (that is, all the processors share a common memory address space). It also introduces the coarray, which allows multiple copies of a program to

run in both shared and distributed memory environments. These are covered in more detail in Section 12.3.

**4. *Write code that can be reused.***

Human time is expensive. Computer hardware and software is cheap in comparison. It is therefore advantageous to write code in such a way that it can be reused in new applications and in new environments with minimal change.

Some well-known techniques that aid reuse include:

- Following the current Fortran standard. The standard is the contract between the compiler writer and the application developer. When nonstandard code is used, there are no guarantees that the code will run with future hardware or software.
- Maximizing the use of local variables. Generally, low-level procedures should both accept their inputs and return their results through the dummy argument list of subroutines and function return values. Use of variables outside the procedure's local scope often creates application-specific dependencies, which can limit reuse.
- Using derived types, and their type-bound procedures and procedure components. These allow code to be reused and even extended, using object-oriented techniques. Components within objects can be added, changed, or removed while limiting the scope of changes in existing code to places that actually use the changed components.

**5. *Document all code changes, keeping a history of all code revisions.***

Auditability refers mostly to the commentary within the code and to its revision history. It is quite useful to understand how and why a particular area changed and to verify that the correct version of a routine is in use. In some environments, such as in organizations that need to conform to Sarbanes-Oxley standards (see Reference [68]), it is critical to maintain a revision history. It is also useful for a program to be able to indicate its own version to a user upon demand.

Source code control systems are very useful for inserting version numbers into source code on a per file basis. For example, in one such system, CVS (see Reference [16]), one can embed the string *$Id: $* into a comment line in the source:

```
! File version: $Id: $
```

The sentinel "Id" is a keyword. When CVS encounters it surrounded by dollar signs, it expands the line by adding a header when extracting the source file. The previous line would expand to something like:

```
! File version: $Id: version_mod.f90,v 1.4 2009/02/02
! 02:55:10 wws Exp $
```

In addition, each change to the file via the source code control system allows the insertion of commentary describing the change. The text is maintained for future review.

A version numbering scheme should also be maintained for the program or library as a whole. Typically these version numbers will use digits separated by periods indicating major, minor, and bug-fix levels. A major release is one where the new features are of such significance that a shift in how a user uses the code may occur. A minor release may signify that features with little impact to existing use have been added and that a large number of bugs have been fixed. Sometimes developers release a very small number of changes purely to fix specific bugs that have been discovered since the previous major or minor releases.

A common convention is to allow a user to specify an option on the command line, such as *-V* or *--version*, that causes the program to print out its version. Programs might also print their version on one of the output files or on the screen. Note that programs can read arguments from the command line with the **get_command_argument** intrinsic procedure.

```
$ prog1 --version
prog1 (Elements utilities) 3.1.4$
```

Additionally, especially in the case of libraries that are used by a number of applications, it is useful to maintain a module variable within the library containing the version string. A module procedure can be written to return the string so that a caller can determine which version of the library is in use. Likewise, the CVS Id can be placed into a character string that can be extracted with tools such as the Unix strings command. By doing this you can ensure that the source file and the object file match.

```
! wws Exp $

module Version_mod
  implicit none
  private
  public :: Check_version_option, Get_version

    ! Because of space requirements, literal constant
    ! CVS_ID is shown on 2 lines. CVS will write it on
    ! one line.

  character(*), parameter :: CVS_ID = &
     '$Id: ProgForAuditability.tex,v 1.16 2010-12-18 &
     &23:26:02 clerman Exp $'
  character(*), parameter :: VERSION_STRING='3.1.4'

contains
```

```
  function Get_version () result (return_value)
    character(len (VERSION_STRING)) :: return_value

    return_value = VERSION_STRING
  end function Get_version

  subroutine Check_version_option ()
    character(128) :: arg_string
    integer :: i

    do, i=1, command_argument_count ()
      call get_command_argument (number=i, &
          value=arg_string)
      if (arg_string == '--version') then
        print *, 'Version: ', Get_version ()
        exit
      end if
    end do
  end subroutine Check_version_option
end module Version_mod
```

# 3.

# Formatting Conventions

## 3.1 Source Form

**6.** ***Always use free source form.***

Fortran 90 introduced free source form. We recommend that it always be used in new code. Free source form offers a number of advantages over the older fixed source form code:

- Free source form is more compatible with modern interactive input devices than fixed form. The maximum line length is 132 characters, compared to the older limit of 72 characters. This reduces the possibility of text exceeding the limit, which could lead the compiler to misinterpret names.
- Line continuations in free form are performed by using a trailing ampersand character, &, rather than entering a character in column 6 of the following line. As an additional visual reminder and safeguard, a leading ampersand, placed in any column, is also allowed to precede the remaining source code.
- In fixed source form, the first six columns are reserved for statement labels, with column 1 also used to indicate comment lines. In modern code, using structured control statements, statement labels are rare. The first five columns are therefore wasted because they are rarely used. These last two features, combined with the next, provide much greater flexibility laying out the code.
- In free source form, any statement can begin in column 1. Free source form always uses the "in-line" comment style, indicated by using an exclamation mark. In-line comments can begin in any column. Here is the same code in fixed format and in free format:

```
C FIXED SOURCE FORM COMMENT
      DO 10, I = 1, SIZE (DTARR)
         ...
   10 CONTINUE


  ! Free format comment
do, i=1, size (dtarr) ! comments begin in any column
  ...
end do
```

- With free source form, the concept of "significant blanks" was introduced. In fixed form source, blanks were insignificant in most contexts. This could lead to code that was very difficult to read. For example, statement

or variable names might be split across line continuations. By requiring blanks to be significant in free form code, code becomes more uniform and readable, leading to better clarity and reliability. Here is a sample of a fixed form statement showing what are now considered significant blanks followed by an equivalent statement without the blanks:

```
DO ITER = 1, MAX ITER S
...
DO ITER = 1, MAXITERS
```

## 3.2 Case

**7. *Adopt and use a consistent set of rules for case.***

It is essential when discussing case in Fortran to emphasize that it is a case-insensitive language. The following variations all represent the same variable: VORTICITY, vorticity, Vorticity, VortiCity. Compilers that, as an optional feature, permit distinguishing entities based solely on case are not standard; you should not use this option.

Strictly speaking, prior to Fortran 90, standard-conforming code had to be written in uppercase letters. The Fortran character set described in the older standards specified only the 26 uppercase letters.

Beginning with Fortran 90, lowercase letters have been formally permitted, and all known compiler vendors supported them. However, because so much old code is still in use, it is still common to encounter code that conforms to the original restriction. Here is a snippet of code from the *LINPACK Users' Guide* (see Reference [20]):

```
20 M = MOD (N, 4)
   IF (M .EQ. 0) GO TO 40
   DO 30 I = 1, M
     DY(I) = DY(I) + DA * DX(I)
30 CONTINUE
   IF (N .LT. 4) RETURN
40 MP1 = M + 1
```

Nowadays, programmers can use lowercase letters, and the underscore is part of the character set. The maximum number of characters that can be used to form a name has grown from the original number of 6, to 31 in Fortran 90/95, to 63 in Fortran 2003. It is now common to see a variable such as molecular_weight.

Even though the language is case-insensitive, you will often see that programs, especially large ones, are written with specific case conventions. Using different combinations of case, such as using all one case for certain types of program entities and the opposite case for others, capitalizing certain types of entities, or using a consistent set of combinations of uppercase letters and underscores,