Appendix A
Introduction to Fortran 2003 via Examples

A

by Gunnar Wollan[1] and Lars Petter Røed

The following gives a brief insight into the Fortran 2003 programming language via specific examples. The reader will learn how to solve a computational problem, and how to handle reading and writing of data to files. For more details, in particular regarding Fortran 90/95, the reader may download Fortran texts from the net. We particularly recommend the site: www.nsc.liu.se/~boein/f90/, which contains versions in both English and Swedish.

A.1 Why Use Fortran?

First, the obvious question: Why should I learn to program in Fortran? In the field of meteorology and oceanography, you will probably come across atmospheric models like WRF and CAM, and ocean models like ROMS, NEMO, or similar. Depending on your project, you will sometimes have to make changes or additions to an already existing model written in Fortran. This task will be decidedly easier if you acquire some knowledge of programming in Fortran. Moreover, valuable time may be lost if you have to acquire that knowledge later on, when you need to make changes to the model.

In the last 15–20 years or so, Fortran has been looked upon as an old-fashioned and poorly structured programming language by researchers and students alike in the field of informatics. The reason is that earlier versions of Fortran lacked most of the features found in modern programming languages like C++, Java, etc. The lack of object orientation has been the main drawback with Fortran. But this is no longer true. Fortran 2003 and Fortran 2008 have all modern features, including object-oriented programming (OOP).

[1] Former scientific programmer at the Department of Geosciences, University of Oslo.

L. P. Røed, *Atmospheres and Oceans on Computers*,
Springer Textbooks in Earth Sciences, Geography and Environment,
https://doi.org/10.1007/978-3-319-93864-6

The most important reason why Fortran is still favored as a programming language within the meteorology and oceanography community is the execution speed of the compiled program. In number crunching speed, Fortran is much faster than C and C++. Tests show that an optimized Fortran program may in some cases run up to 30 percent faster than the equivalent C or C++ program. Thus, for large and complex programs and codes with a runtime of weeks, even a small increase in speed will significantly reduce the overall time it takes to solve a problem. This is important in the field of meteorology and oceanography, since speed is everything when producing a forecast. In addition, laboratory experiments and fieldwork are sometimes costly to perform. Computer simulations are less costly and are therefore becoming increasingly important as an addition to laboratory and fieldwork. Even then speed is a factor.

A.2 Historical Background

Fortran is indeed an old programming language. As early as 1954, John W. Backus[2] and his team at IBM began developing the scientific programming language Fortran. It was first introduced in 1957 for a limited set of computer architectures. After a short time, the language had spread to other architectures. Since then, it has been the most widely used programming language for solving numerical problems in the natural sciences in general and in atmosphere and ocean science in particular.

The name Fortran is derived from Formula Translation and, as already mentioned, is still the language of choice for fast numerical computations. In 1959, a new version, Fortran II, was introduced. This version was more advanced, and among the new features it could use complex numbers and split a program into subroutines. In the years to follow, Fortran was further developed to become a programming language that was fairly easy to understand and well adapted to solve numerical problems.

In 1962, a new version, Fortran IV, emerged. Among its new features was the ability to read and write direct access files. In addition, it introduced a new data type called `LOGICAL`. This was a Boolean data type with two states *true* or *false*. At the end of the 1970s, Fortran 77 was introduced. This version contained better loop and test structures. In 1992, Fortran 90 was formally introduced as an ANSI/ISO standard, followed shortly afterwards by Fortran 95, a minor extension of Fortran 90. These versions turned Fortran into a modern programming language and included many of the features expected of a modern programming language. Finally, Fortran 2003 was released, incorporating even OOP with type extension and inheritance, polymorphism, dynamic type allocation, and type-bound procedures.

[2] John Warner Backus (1924–2007) was an American computer scientist. He directed the team that invented the first widely used high-level programming language (FORTRAN) and was the inventor of the Backus–Naur form, a widely used notation to define formal language syntax. He also did research in function-level programming and helped to popularize it. The IEEE awarded Backus the W.W. McDowell Award in 1967 for the development of FORTRAN. He received the National Medal of Science in 1975 (Source: Wikipedia).

A.3 Fortran Syntax

The starting point is that Fortran, like all programming languages, has its own syntax. To start programming in Fortran, a knowledge of its syntax is therefore required. In Fortran, as well as other programming languages, the code is separated into a part declaring variables through declaration statements, and a part carrying out instructions for manipulating the contents of the variables. An important difference between the earlier Fortran 77 and Fortran 90/95/2003 is the way in which each separate line of code is written.

In Fortran 77, it was written in fixed form, where each line of code was divided into 80 columns, and each column had its own meaning. This division had a historical background. In the 1960s and part of the 1970s, the standard medium for data input was punched cards like the one shown in Fig. A.1. The cards were divided into 80 columns and it was therefore natural to set the length of each line of code to 80 characters. Table A.1 gives an overview of the subdivision of the line of code. Note that Fortran 77 is a subset of Fortran 2003. Thus, all programs written in Fortran 77 can be compiled using a Fortran 2003 compiler.

In addition to the fixed code format from Fortran 77, Fortran 2003 also supports free format coding. This means that the separation into columns is no longer necessary and the program code can be written in a more structured way. This makes it more readable and far easier to maintain. Today, *the free format is the default* setting for Fortran 90/95 and Fortran 2003 compilers.

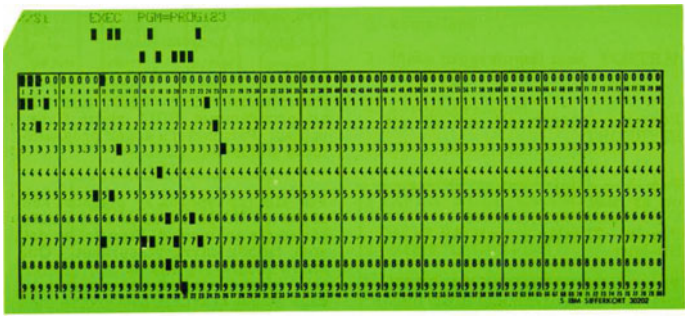

Fig. A.1 The punched card

Table A.1 Fortran 77 (F77) fixed format

Column	Meaning
1	A character here means the line is a comment
2–5	Jump address and format number
6	A character here is a continuation from the previous line
7–72	Program code
73–80	Comment

A.3.1 Data Types in Fortran

In earlier versions of Fortran, four basic data types were included. These were `INTEGER` and `REAL` numbers, a Boolean type called `LOGICAL`, and `CHARACTER`, which represents the alphabet and other special non-numeric types. In Fortran 90/95, the `REAL` data type is split into the `REAL` and `COMPLEX` data types. In addition to this, a derived data type can be used in Fortran 2003. A derived data type may contain one or more of the basic data types, other derived data types, and in addition, procedures which are part of the new OOP features in Fortran 2003.

`INTEGER`

An `INTEGER` data type is identified with the reserved word `INTEGER`. It has a valid range which varies with the way it is declared and the architecture of the computer it is compiled on. When nothing else is given, an `INTEGER` has a length of 32 bits on a typical workstation and can have a value from -2^{31} to 2^{30}, while a 64-bit `INTEGER` has a minimum value of -2^{63} and a maximum value of 2^{62}.

`REAL` and `COMPLEX`

In the same manner, a `REAL` number can be specified with various ranges and accuracies. A real number is identified with the reserved word `REAL` and can be declared with single or double precision. Table A.2 gives the number of bits and minimum and maximum values. A double precision real number is declared using the reserved words `DOUBLE PRECISION` or `REAL(KIND=8)`. The latter is the preferred declaration.

An extension of `REAL` numbers is `COMPLEX` numbers with their real and imaginary parts. A `COMPLEX` number is identified with the reserved word `COMPLEX`. The real part can be extracted by the function `REAL()` and the imaginary part by the function `AIMAG()`. There is no need to write explicit calculation functions for `COMPLEX` numbers, as one has to do in C/C++, which lacks the `COMPLEX` data type.

`LOGICAL`

The Boolean data type is identified by the reserved word `LOGICAL` and has only two values: "true" or "false". These values are identified with `.TRUE.` or `.FALSE.`. The dot (period mark) at the beginning and end of the declaration is a necessary part of the syntax. To omit one or more dots will give a compilation error.

`CHARACTER`

The `CHARACTER` data type is identified by the reserved word `CHARACTER` and contains letters and characters in order to represent data in a readable form. Legal characters are, among others, `a` to `z`, `A` to `Z`, and some special characters `+`, `-`, `*`, `/` and `=`.

Table A.2 The `REAL` numbers data type in Fortran

Precision	Sign	Exponent	Significand	Maximum value	Minimum value
Single	1	8	23	2^{128}	2^{-126}
Double	1	11	52	2^{1024}	2^{-1022}

`Derived Data Types`
These are data types which are defined for special purposes. A derived data type is built from components from one or more of the four basic data types, and also other derived data types. A derived data type is always identified by the reserved word `TYPE name` as prefix and `END TYPE name` as postfix.

A.4 Structure of Fortran

A.4.1 Declaration of Variables

In Fortran, there are two ways to declare a variable. The first is called *implicit* declaration and is inherited from the earliest versions of Fortran. The second is called *explicit* declaration and is in accordance with other programming languages. Explicit declaration means that all variables have to be declared *before* any instruction occurs.

Implicit declaration, on the other hand, means that a variable is declared when needed by giving it a value anywhere in the source code, that is, even within the instructions. The data type is determined by the first letter in the variable name. An `INTEGER` is recognized by starting with the letters `I` to `N` and a `REAL` variable by the rest of the alphabet. No special characters are allowed in a variable name. Only the letters `A–Z`, the numbers `0–9`, and the underscore character _ are allowed. A variable cannot start with a number. In addition, a `LOGICAL` variable is, in most compilers, identified by the letter `L`.

It should be emphasized, as a general rule, that an implicit declaration is not a good way to program. For one thing, it renders a code that is not easy to read. For another, it easily introduces errors into a program due to the risk of typing errors. It is therefore strongly recommended to use explicit declaration of variables. To ensure that all variables are declared, the keywords `IMPLICIT NONE` must be included in the second line of all programs, functions, and subroutines. This tells the compiler to check that all variables are declared. Finally, there are some variables that always have to be declared. These are arrays in one or more dimensions and character strings.

`INTEGER` Numbers
A good start is to show an example of how to declare an `INTEGER` in Fortran 95:

```
INTEGER                 :: i ! Declaration of an INTEGER
                            ! length(32 bit)
INTEGER(KIND=2)         :: j ! Declaration of an INTEGER (16bit)
INTEGER(KIND=4)         :: k ! Declaration of an INTEGER (32bit)
INTEGER(KIND=8)         :: m ! Declaration of an INTEGER (64bit)
INTEGER,DIMENSION(100):: n ! Declaration of an INTEGER array
                            ! (100 elements)
```

There are certain differences between the Fortran 77 and Fortran 95 ways of declaring variables. In Fortran 95 there is more to write, but this is offset by greater readability.

Note that in Fortran 95 a comment can start anywhere on the code line, but must always be preceded by an exclamation (!) mark.

REAL Numbers

In most compilers, the REAL data type now conforms to the IEEE standard for floating point numbers. Declarations of single and double precision are declared as in the following example:

```
REAL                :: x ! Declaration of REAL
                         ! defaultlength (32bit)
REAL(KIND=8)        :: y ! Declaration of REAL
                         ! double precision (64 bit)
REAL,DIMENSION(200) :: z ! Declaration of REAL array
                         ! (200 elements)
```

COMPLEX Numbers

Unlike C/C++, Fortran has an intrinsic data type of complex numbers. Declaration of a COMPLEX variable in Fortran is achieved as follows:

```
COMPLEX                :: a ! Complex number
COMPLEX,DIMENSION(100) :: b ! Array of complex numbers
                            ! (100 elements)
```

LOGICAL Variables

Unlike INTEGER and REAL numbers, a LOGICAL variable has only two values, .TRUE. or .FALSE., and therefore uses a minimum of space. The number of bits a LOGICAL variable uses depends on the architecture and the compiler. It is possible to declare a single LOGICAL variable or an array of them. The following example shows a Fortran 90/95 declaration. In other programming languages, the LOGICAL variable is often called a Boolean variable, after the mathematician George Boole.[3]

```
LOGICAL                :: l1 ! Single LOGICAL variable
LOGICAL,DIMENSION(100) :: l2 ! Array of LOGICAL variables
                             ! (100 elements)
```

CHARACTER Variables

Characters can either be declared as a single CHARACTER variable, a string of characters, or an array of single characters or character strings:

[3] George Boole (1815–1864) was an English mathematician, philosopher, and logician. He worked in the fields of differential equations and algebraic logic, and is now best known as the author of "The Laws of Thought", and as the inventor of the prototype of what is now called Boolean logic (source: Wikipedia).

```
CHARACTER                          :: c1 ! Single character
CHARACTER(LEN=80)                  :: c2 ! String of characters
CHARACTER,DIMENSION(10)            :: c3 ! Array of single
                                         ! characters
CHARACTER(LEN=80),DIMENSION(10) :: c4 ! Array of character
                                         ! strings (10 elements)
```

Derived Data Types
The Fortran 95 syntax for the declaration of a derived data type can be like the one shown here:

```
TYPE derived
   ! Internal variables
   INTEGER           :: counter
   REAL              :: number
   LOGICAL           :: used
   CHARACTER(LEN=10) :: string
END TYPE derived
! A declaration of a variable of
! the new derived data type
TYPE (derived)    :: my_type
```

The question arises as to why we should use derived data types. The answer is that it is sometimes desirable to group variables together to be able to refer to them under a common name. It is usually good practice to select a name of abstract data type to indicate the contents and area of use.

A.4.2 Instructions

There are two main types of instructions. One is for program control and the other is for assigning a value to a variable.

Instructions for Program Control
Instructions for program control can be split into three groups, one for loops, a second for tests (even though a loop usually involves an implicit test), and a third for assigning values to variables and performing mathematical operations on the variables.

In Fortran, all loops start with the reserved word `DO`. The following piece of code shows a short example of a simple loop:

```
DO i = 1, 100
  !// Here instructions are performed 100 times
  !// before the loop is finished
END DO
```

The next example shows a loop with instructions. This loop is a nonterminating loop, where an `IF`-test inside the loop is used to exit the loop when the result of the test is true:

```
....
do
  a = a * sqrt(b) + c
  if (a > z) then
    !// Jump out of the loop
    exit
  end if
end do
```

This piece of code instructs the computer to give the variable `a` a value equal to the sum of the square root of the variable `b` multiplied by the previous value of `a` and a third variable `c`. When the value of `a` becomes greater then the value of the variable `z`, the program transfers control to the next instruction following the loop. Note that it is assumed that all the variables are declared and initialized somewhere in the program *before* the loop, as indicated by the code line `....` appearing before the loop. The various Fortran instructions will be described in the example in Sect. A.6.

A.5 Compiling a Program

In order to have an executable program, it must be compiled. This requires the sample program (or source code) to reside in a file on the computer, say `daynr.f90`, where the extension indicates that the file contains a Fortran program written in Fortran 90. The compilation process takes the file with the source code and creates a binary file linked in with the necessary system libraries, so that it may run the program on the computer. The binary file contains the program in machine-specific assembly language, i.e., instructions written in machine language.

The most common compiler is the open-source compiler called `gfortran`.[4] The command line for compiling the program `daynr.f90` is simply

```
gfortran -o daynr daynr.f90
```

where `gfortran` is the name of the compiler. The argument `-o` means that the next argument to the compiler is the name of the executable program. The last argument is the name of the file containing the source code. One may also simply write

```
gfortran daynr.f90
```

In this case, the executable program by default is given the name `a.out`. To run the compiled program, use the command `./daynr` (or `./a.out`) in the terminal window.

[4] Open source means that the compiler may be downloaded and used free of charge.

A.6 Sample Programs

A.6.1 Daynumber Converter

The first sample program considered is a very simple one in which the task is to calculate the daynumber of a specific date in the year. It is assumed that the year is a non-leap year. We first write the program skeleton, and then fill in the necessary code to solve the problem:

```
PROGRAM daynumber
  implicit none

END PROGRAM daynumber
```

All Fortran programs begin with the reserved word `PROGRAM`, followed by the program name. In this case, the program name is `daynumber`. As mentioned above, the code line `implicit none` is almost mandatory, or at least good programming practice. It prevents the use of implicit declarations, which are otherwise the default behavior of the Fortran compiler.

Some variables and constants are then declared to calculate the daynumber:

```
PROGRAM daynumber
  implicit none
  integer                :: counter
  integer,dimension(12)  :: months
  integer                :: day, month
  integer                :: daynr

END PROGRAM daynumber
```

Four integer variables are declared, namely, `counter`, `day`, `month`, and `daynr`, and one integer array `months` with 12 elements. The variable `counter` is used to traverse the array to select the number of days in the months before the given month. The variables `day` and `month` hold the day and month. The variable `daynr` contains the result of the calculations.

Next, numbers are specified for the constant integers `day` and `month`, and the variable `daynr` and the array `months` are initialized:

```
PROGRAM daynumber
  implicit none

  day = 16
  month = 9
  daynr = 0
  months(:) = 31

END PROGRAM daynumber
```

Initializing scalar arrays is not difficult, but each element of the array usually has to be initialized separately. Fortunately, Fortran 95 and 2003 have a built-in functionality which allows initialization of a whole array with one value.

However, not all months contain 31 days. Thus, the next step is to change the number of days in the months that differ from 31, that is, `months(2)` (February), `months(4)` (April) , `months(6)` (June), `months(9)` (September), and `months(11)` (November):

```
PROGRAM daynumber
  implicit none

  months(2) = 28
  months(4) = 30
  months(6) = 30
  months(9) = 30
  months(11) = 30

END PROGRAM daynumber
```

The next step is to loop through all the elements in the array `months` up to the month minus one, summing up the number of days in each month in the `daynr` variable. To arrive at the result, the value from the variable day is added to `daynr`. To display the result, the command `PRINT *, daynr` writes the result on the terminal:

```
PROGRAM daynumber
  implicit none

  DO counter = 1, month - 1
    daynr = daynr + months(counter)
  END DO
  daynr = daynr + day
  PRINT*, daynr
END PROGRAM daynumber
```

The resulting output from this sample program with `month` = 9 and `day` = 16 is 259. You can use a calculator and perform the calculations by hand to check that the result is correct.

This simple program illustrates the rule that one should *never use implicit declarations of variables*. This is very important. There is a story from the 1970s about 10 implicit declarations where a typing error created an uninitialized variable that caused a NASA rocket launch to fail, and the rocket had to be destroyed before it could cause serious damage.

A.6.2 Temperature Converter

The next sample program we consider converts a temperature from degrees Fahrenheit to degrees Celsius (or centigrade). The two temperatures must appear side by side in the terminal window. The formulas for the conversions are

$$C = \frac{5}{9}(F - 32) \,, \qquad F = \frac{9}{5}C + 32 \,, \tag{A.1}$$

where F represents the temperature in Fahrenheit and C the temperature in Celsius or centigrade.

To proceed, two floating point variables are needed, one to hold the temperature in Fahrenheit, say `F`, and a second to hold the temperature in Celsius, say `C`. To declare them as floating point variables, we use the `REAL` keyword. Thus, the following piece of code must be included:

```
!// The Fahrenheit variable
REAL :: F
!// The Centigrade variable
REAL :: C
```

Note that, in contrast to most other languages, Fortran is case insensitive. That means that a variable or function name is the same whether it is written with uppercase or lowercase letters. Moreover, beginning with the Fortran 90 version, a double colon is used to separate the variable type from the variable name, while an exclamation sign is used to tell the compiler that the rest of the line is a comment. In order to make the code more readable, it is recommended to include an additional double slash before writing the comment, as shown in the above example. A program that is easy to read is also easy to understand, and this makes it easier to find and correct errors.

The next step is to initialize the Fahrenheit variable and perform the conversion according to the formula. The whole program may be something like this:

```
PROGRAM f2c_simple
  IMPLICIT NONE
  !// Declare variables
  REAL :: F    ! Fahrenheit variable F (floating point)
  REAL :: C    ! Centigrade variable C (floating point)
  !// Assign a value to F as a constant number
  !// Note the decimal point
  F = 75.
  !// Perform the calculations
  C = (F - 32.)*5./9.
  !// Write the result to the terminal window
  PRINT *, C
END PROGRAM f2c_simple
```

The Fortran default of implicit declarations of variables is avoided by writing `IMPLICIT NONE` in the second code line. To tell the compiler that the constant value 75 is a real number, a decimal point is added as part of the number. If the

dot is omitted the compiler will assume that it is an integer, and in some cases, the calculations will be wrong. Finally, parentheses are there to ensure that calculations are performed in the right order. The statement or command `PRINT *, C` writes the temperature in degrees Celsius in the terminal window.

As in the previous example, the program has to be compiled to obtain a binary executable program file. There are several commercial Fortran compilers, but the open-source GNU Fortran compiler `gfortran` is used in this example to compile the program. Assuming that the program is written into a file named `f2c_simple.f90`, it is compiled by the command

```
gfortran -o f2c f2c_simple.f90
```

where the `-o` option tells the compiler that the next argument is the name of the executable program and the last argument is the name of the file with the source code. In this case, the binary program file `f2c` is created. This may be run by typing `./f2c` in the terminal window. This is exactly as for the sample program in Sect. A.6.1. All Fortran programs are compiled and run this way.

A.6.3 A More User-Friendly Version of the Converter Program

The above program is not very user friendly. Every time a new temperature in degrees Fahrenheit is to be converted to degrees Celsius, the source code has to be changed. A new `F` has to be specified, and then the program has to be recompiled and rerun. To avoid this, a user interface asking for a temperature in degrees Fahrenheit may be added to the program.

This is accomplished by adding some code lines for communication in the form of text strings. In the example below, two text strings `prompt1` and `prompt2` are declared and assigned. The first text string, declared as `prompt1`, asks us whether our input is in Fahrenheit or Celsius, while the second, `prompt2`, asks us to enter the temperature that we, the user, would like to convert. To declare the text strings, the data type `CHARACTER` is used. Thus, the program begins with the following code lines:

```
!/////////////////////////////////////////////////////
!//
!// f2c.f90
!//
!// Program to convert from Fahrenheit to
!// Celsius or vice versa
!/////////////////////////////////////////////////////
PROGRAM f2c
  IMPLICIT NONE
  REAL :: F       !// Temperature in Fahrenheit
  REAL :: C       !// Temperature in degrees Celsius
  !// Character strings to hold the prompts for
  !// communicating with the user
```

```
CHARACTER(LEN=80) :: prompt1, prompt2
!// A single character to hold the answer
!// which is either F or C
CHARACTER :: answer
!// Assign a value to prompt1
prompt1 = 'Enter F for Fahrenheit or C for Celsius'
!// Assign a value to prompt2
prompt2 = 'Enter a temperature'
```

The declaration `CHARACTER(LEN=80)` means that a space for a text string up to 80 characters long is allocated. The character `answer` is unassigned, but is used to hold a single character variable that is the answer to the question `prompt1`, that is, `answer` is used to hold the characters (`F` or `C`) according to the answer to `prompt1`.

Note also that some comments are added above the `PROGRAM f2c` line. They give the name of the file containing the source code, and also a brief description of the purpose of the program. This makes the code more readable and easier to understand and is recommendable, even for short programs like this one.

The code continues:

```
!// Print the contents to the terminal window
!// without trailing blank characters
PRINT *, TRIM(prompt1)
!// Read the input from the keyboard
READ(*,*) answer
!// Print the contents to the terminal window
!// without trailing blank characters
PRINT *, TRIM(prompt2)
```

This part prints the assigned value of `prompt1` to the terminal window, and then reads the input and puts it in `answer`. It subsequently prints `prompt2` to the terminal window and waits for the next input. The keyword TRIM tells the compiler to print out the text without printing the trailing space characters. To read the answer to `prompt1`, the `READ` command is used. The construct `READ(*,*)` instructs the computer to read the input from the keyboard into the receiving variable using the declared format for that variable.

The next program step is:

```
!// Is the temperature given in Fahrenheit?
IF(answer .EQ. 'F') THEN
  !// Yes, read input into the F variable
  READ(*,*) F
  !// Convert from Fahrenheit to Celsius
  C = (F - 32) * (5. / 9.)
  !// Print the result to the screen
  PRINT *, C
ELSE
  !// No, read input into the C variable
  READ(*,*) C
```

```
      !// convert from Celsius to Fahrenheit
      F = (C * 9. / 5.) + 32
      !// Print the result to the screen
      PRINT *, F
    END IF
END PROGRAM f2c
```

This reads the input, then branches out in accordance with the input `answer`, performs the conversion, and prints the result before ending the program properly using the code line `END PROGRAM f2c`.

A.6.4 Variable Types, Arrays, Loops, and Memory Allocation

More often than not, atmosphere and ocean variables are stored in large files where the variables are stored in multiple dimensional arrays. To illustrate this, it is best to begin by looking at the way data are stored in a one-dimensional array.[5]

For this purpose, we construct a program that calculates the Fibonacci sequence.[6] The sequence consists of a series of integers, the so-called Fibonacci numbers. These must be stored in a one-dimensional array or vector of integers. Recall that the formula for calculating the Fibonacci sequence is

$$F_{j-1} = F_{j-2} + F_{j-3}\ , \quad j = 3(1)n\ , \tag{A.2}$$

where the two first numbers are $F_0 = 0$ ($j = 1$) and $F_1 = 1$ ($j = 2$).

First, we require an array of integers to hold the numbers. The program will be constructed so that the user may choose the length of the sequence. Thus, we must include a user interface like the one in the temperature conversion program, asking for the length of the sequence. The length of this array (or vector) is not known in advance. We must therefore use what is known as an allocatable array, where the needed space is allocated at runtime. In addition, an index variable and a status variable are needed. The first is needed to access the various elements of the array, and the second to check the result of the allocation. Consequently, the first part of the code takes the form:

```
!////////////////////////////////////////////////////////
!//
!// fibonacci.f90
!//
```

[5] A one-dimensional array is often referred to as a vector.

[6] Leonardo Pisano Bigollo (ca. 1170–ca. 1250), also known as Fibonacci, Leonardo of Pisa, Leonardo Pisano, Leonardo Bonacci, and Leonardo Fibonacci, was an Italian mathematician, considered by some "the most talented Western mathematician of the Middle Ages". Fibonacci is best known to the modern world for spreading the Hindu–Arabic numeral system in Europe, primarily through his Liber Abaci (Book of Calculation), written in 1202, and for a numerical sequence called the Fibonacci sequence, which he did not discover, but used as an example in the Liber Abaci.

```
!// Program to display the Fibonacci
!// sequence from 1 to n
!//
!////////////////////////////////////////////////////////
PROGRAM fibonacci
  IMPLICIT NONE
!// Variable declarations
 !// Declaring integer variables
  !// first a counter variable
  INTEGER :: i
  !// then the length of the Fibonacci sequence
  INTEGER :: n
  !// and finally a status variable (if errors)
  INTEGER :: res
  !// Declare an array to hold the Fibonacci sequence
  !// with unknown length at compilation time
  INTEGER, ALLOCATABLE, DIMENSION(:) :: sequence
```

To declare an array with unknown length, in this case `sequence`, the keywords `INTEGER, ALLOCATABLE, DIMENSION(:)` are used, where the length of the array is replaced by a colon `:`. If the length of the array is known in advance, say an array containing 100 elements, the construct `INTEGER, DIMENSION(100)` is used.

The next step is to prompt for how many numbers of the infinite Fibonacci sequence are to be included in the calculation. Later this number is used to allocate space for the Fibonacci numbers in the sequence. Thus, a prompting character string of a certain length, say 80, has to be declared and assigned a value. As long as the text string is shorter than 80 characters, a line of code counting the actual number of characters is inserted. Furthermore, the prompt is pushed to the terminal window. Thus, the next part of the program takes the form

```
!// Declare the length of the prompt to ask
!// for the length of the Fibonacci sequence
CHARACTER(LEN=80) :: prompt
!// Assign value to the prompt
prompt = 'Enter the length of the sequence: '
!// Get the number of non blank characters in the prompt
i = LEN(TRIM(prompt))
!// Display the prompt asking for the length suppressing
!// the line feed
WRITE(*,FMT='(A)',ADVANCE='NO') prompt(1:i+1)
```

Since the keyword `PRINT *` always prints the text to the terminal window with a linefeed as the last operation, it is replaced by the command

```
WRITE(*,FMT='(A)',ADVANCE='NO')
```

The `WRITE` command makes it possible to avoid or suppress the linefeed by adding the formatting code in the `WRITE` command, as shown above.

To read in the input from the terminal window and then perform the calculations, the rest of the program takes the form:

```
  !// Read the keyboard input
  READ(*,*) n
  !// Allocate space for the sequence
  ALLOCATE(sequence(n), STAT=res)
  !// Test the value of the res variable for errors
  IF(res /= 0) THEN
    !// We have an error. Print a message and stop the program
    PRINT *, 'Error in allocating space, status: ', res
    STOP
  END IF
  !// Initialize the two first elements in the sequence
  sequence(1) = 0
  sequence(2) = 1
  !// Loop and calculate the Fibonacci numbers
  DO i = 3, n
    sequence(i) = sequence(i-1) + sequence(i-2)
  END DO
  !// Print the sequence to the screen
  PRINT *, sequence
END PROGRAM fibonacci
```

Once the variable `n` has been read, space is allocated for the Fibonacci array `sequence` using the construct

```
ALLOCATE(sequence(n), STAT=res)
```

Then, using the integer `res`, an "if" test is made to check whether the allocation is acceptable. If different from zero, the allocation has failed and the program is stopped. Otherwise, the program continues by first specifying the first two numbers in the Fibonacci sequence. Next, a loop is started to calculate the following integers in the sequence using the formula given in (A.2). In accordance with (A.2), the loop starts with an index variable equal to 3. Finally, the code line `PRINT *, sequence` is added at the end to display the contents of the sequence in the terminal window (unformatted).

A.6.5 File Input/Output or I/O

The data input used in the programs is usually stored in files. These may be numbers generated through the output of another program, or observations produced by instrument sensors in one way or another. In either case, they are usually available to us on a file stored on a computer somewhere, or residing on a memory device of some sort.

In the example to follow, the reader will learn how to read data from a file into an array, perform some operation on the data set, and write the result to a new file. It is

assumed that the length of the array, say seven elements long, is known in advance. The program starts like this:

```
!////////////////////////////////////////////////////////
!//
!// f2c_file.f90
!//
!// Program to calculate the degrees Celsius from a
!// file containing seven observations of temperatures
!// in Fahrenheit
!//
!////////////////////////////////////////////////////////
PROGRAM f2c_file
  IMPLICIT NONE
  !// Declare a static variable to hold the
  !// length of the arrays
  INTEGER, PARAMETER :: n = 7
  !// Declare the arrays for the temperatures,
  !// and a counter variable
  REAL,DIMENSION(n)  :: F             !// Fahrenheit
  REAL,DIMENSION(n)  :: C             !// Celsius
  INTEGER            :: j             !// Counter variable
```

To specify that `n` is constant or static, it is declared using the keyword `INTEGER, PARAMETER` to signal that it is unchanged at runtime. Next, the arrays that will hold the temperatures in Fahrenheit and Celsius are declared together with a counter variable. In addition to this, two character strings are needed to hold the names of the input and output files, as given by the following lines of code:

```
  !// Declare character strings to hold the filenames
  CHARACTER(LEN=80)  :: infile  !// Holds the Fahrenheit
                                !// temperatures
  CHARACTER(LEN=80)  :: outfile !// Holds the results of
                                !// the conversion to
                                !// degrees Celsius
```

Furthermore, two parameters are needed to hold the unit numbers, which are used to reference the input and output files. The names of the input and output files are also needed, to recognize them in the directory once the operation is completed. The reference numbers are integers. In this regard, a status variable must be declared, as in the last example above. Thus, the next part of the program takes the form:

```
  !// Declare constant values for the Logical Unit Number for
  !// referencing the files for opening, reading, and writing
  INTEGER, PARAMETER :: ilun = 10
  INTEGER, PARAMETER :: olun = 11
  !// A status variable to hold the result of file
  !// operations
  INTEGER :: res
```

```
!// Assign an input filename for temperatures in Fahrenheit
infile = "fahrenheit.txt"
!// Assign an output filename for temperatures in Celsius
outfile = "celsius.txt"
```

To access the contents of the output file, it has to be opened using the declared unit number and its filename `infile`. We also add a test to see whether the opening was successful. To achieve this, the program continues with the following statements:

```
!// Open the input file
OPEN(UNIT=ilun,FILE=infile,FORM="FORMATTED",IOSTAT=res)
!// Test whether the operation was successful
IF(res /= 0) THEN
  !// No, an error occurred, print a message to
  !// the screen
  PRINT *, "Error in opening file, status: ", res
  !// Stop the program
  STOP
END IF
```

The next step is to read the contents of the input file into the array `F`. This is carried out using a loop that runs from `1` to the length of the array, in this example `n`. This is done using the `READ` statement, which is similar to the `OPEN` statement:

```
!// Loop and read each line of the file
DO j = 1, n
  !// Read the current line
  READ(UNIT=ilun,FMT='(F4.1,X,F4.1)',IOSTAT=res) F(j)
  !// Was the read successful?
  IF(res /= 0) THEN
    !// No, Test whether we have reached End Of File (EOF)
    IF(res /= -1) THEN
      !// No, an error has occurred, print a message
      PRINT *, "Error in reading file, status: ", res
      !// Close the file
      CLOSE(UNIT=ilun)
      !// Stop the program
      STOP
    END IF
  END IF
END DO
```

The `infile` is formatted, so we need to specify the format of the data contained in it. In this example, it is the file `fahrenheit.txt`. The argument specified in `FMT` must be an exact match of the format in `fahrenheit.txt`. The keyword/argument pair `FMT='(F4.1,X,F4.1)'` is used to tell the computer that the number is a floating point number with four digits, including the decimal point and one decimal, a space character, and then another floating point number like the first.

The conversion from Fahrenheit to Celsius may then proceed, storing the result in the second array, that is, C. To see the result in the terminal window, we also add a PRINT statement. This is accomplished by adding the code lines:

```
!// Loop and convert from Fahrenheit to Celsius
DO j = 1, n
  C(j) = (F(j) - 32) * 5. / 9.
END DO
!// Print the temperatures to the screen
DO j = 1, n
  PRINT *, " Degrees Fahrenheit ", F(j), &
           " Degrees Celsius ", C(j)
END DO
```

Once the conversion is completed, the contents of the array C may be written to the output file outfile, which was given the reference number olun = 11. To enable writing to the output file, it must be opened. This is carried out exactly as was done for the input file:

```
!// Open the output file
OPEN(UNIT=olun,FILE=outfile,FORM="FORMATTED",IOSTAT=res)
!// Test whether the operation was successful
IF(res /= 0) THEN
  !// No, an error occurred, print a message to
  !// the screen
  PRINT *, "Error in opening output file, status: ", res
  !// Stop the program
 STOP
END IF
```

To ensure that the outfile is formatted, the argument FORM="FORMATTED" is included in the OPEN statement.

The result of the conversion may now be written to the output file, by continuing with the following code lines:

```
DO j = 1, n
  WRITE(UNIT=olun,FMT='(F4.1,A1,F4.1)',IOSTAT=res) C(j)
  !// Test whether the operation was successful
  IF(res /= 0) THEN
    !// No, an error occurred, print a message to
    !// the screen
    PRINT *, "Error in writing file, status: ", res
    !// Exit the loop
    EXIT
  END IF
END DO
```

Finally, the input and output files must be closed before terminating the program:

```
    !// Close the input file
    CLOSE(UNIT=ilun)
    !// Close the output file
    CLOSE(UNIT=olun)
END PROGRAM f2c_file
```

When using the `OPEN` function, use was made of the respective unit numbers (or logical unit numbers) `ilun = 10` and `olun = 11`. As part of the `FILE` argument, the filename was also provided. Finally, note that the result of the call to `OPEN` is returned in the `IOSTAT=res` keyword/argument pair. The same procedure may be used in reading from files.

In this example, formatted files were used. This means that the file content may be displayed in the terminal window. In contrast, writing binary files to the terminal is not very meaningful. To visualize, consider constructing a formatted file containing two pairs of seven temperatures in Fahrenheit, in two columns, formatted following the keyword/argument `FMT='FMT='(F4.1,X,F4.1)'`:

```
68.2 65.5
69.6 63.7
73.2 66.0
75.0 68.0
77.5 70.2
79.2 71.4
91.2 73.2
```

In contrast, the binary file looks like this:

```
The binary file looks like this:
.....
^@H<8D><BC>$<D0>^@^@^@<BE>
^@^@^@<B9><B8>_N^@H<C7>D$0P^@^@^@<BA>^C<FF><84>
^C3<C0>H<C7>D$8<80><DC>s^@L<8D>D$0H<C7>D$@
^@^@^@H<C7>D$H<E0>'N^@H<C7>D$P^D^@^@^@<E8>O<C3>
.....
This is the end of the binary file$
```

Clearly, the binary format is unreadable as is, unless a program is used to translate the binary information into a text file written in ASCII (American Standard Code for Information Interchange) format. ASCII is the format used by formatted files in Fortran.

A.6.6 Multidimensional Arrays

In the field of meteorology and oceanography, data sets are usually in four dimensions, three in space and one in time. Consequently, matrices in four dimensions have to be declared to be able to store the data. The next example shows how the entries

in a two-dimensional matrix, consisting of temperatures in degrees Fahrenheit from two observing stations, are read in from a file, converted to degrees Celsius, and finally written to a file.

The program is of course very similar to the previous example, except that the operation is on a two-dimensional matrix. This is made possible using nested loops. The complete code is:

```
!////////////////////////////////////////////////////////////
!//
!// f2c_advanced.f90
!//
!// A program calculating degrees Celsius from Fahrenheit
!// from two measuring stations
!//
!////////////////////////////////////////////////////////////
PROGRAM f2c_advanced
  IMPLICIT NONE
  !// Declare a static variable to hold the
  !// dimension of the vectors or arrays
  INTEGER, PARAMETER :: m = 7
  INTEGER, PARAMETER :: n = 2
  !// Declare the arrays for the temperatures in
  !// Fahrenheit and Celsius, and counter variables
  REAL,DIMENSION(m,n) :: F
  REAL,DIMENSION(m,n) :: C
  INTEGER :: j, k
  !// Character strings to hold the filenames
  CHARACTER(LEN=80) :: F_file
  CHARACTER(LEN=80) :: C_file
  !// Constant values for the Logical Unit Number
  !// for referencing the files for opening, reading,
  !// and writing
  INTEGER, PARAMETER :: ilun = 10
  INTEGER, PARAMETER :: olun = 11
  !// A status variable to hold the result of file
  !// operations
  INTEGER :: res
  !// Assign the input filename for Fahrenheit temp.
  infile = "fahrenheit.txt"
  !// Assign the output filename for Centigrade temp.
  outfile = "celsius.txt"
  !// Open the Fahrenheit input file
  OPEN(UNIT=ilun,FILE=infile,FORM="FORMATTED",IOSTAT=res)
  !// Test whether the operation was successful
  IF(res /= 0) THEN
    !// No, an error occurred, print a message to
    !// the screen
    PRINT *, "Error in opening file, status: ", res
    !// Stop the program
    STOP
```

```
  END IF
  !// Loop and read each line of the file
  DO j = 1, m
    !// Read the current line
    READ(UNIT=ilun,FMT='(F4.1,X,F4.1)',IOSTAT=res) F(j,1), F(j,2)
    !// Successfully read?
    IF(res /= 0) THEN
      !// No, Test whether we have reached End Of File (EOF)
      IF(res /= -1) THEN
        !// No, an error has occurred, print a message
        PRINT *, "Error in reading file, status: ", res
        !// Close the file
        CLOSE(UNIT=ilun)
        !// Stop the program
        STOP
      END IF
    END IF
  END DO
  !// Close the input file
  CLOSE(UNIT=ilun)
  !// Loop and convert from Fahrenheit to Centigrade
  DO k = 1, n
    DO j = 1, m
      C(j,k) = (F(j,k) - 32.) * 5./9.
    END DO
  END DO
  !// Open the Centigrade output file
  OPEN(UNIT=olun,FILE=centigradefile,FORM="FORMATTED",IOSTAT=res)
  !// Test whether the operation was successful
  IF(res /= 0) THEN
    !// No, an error occurred, print a message to
    !// the screen
    PRINT *, "Error in opening output file, status: ", res
    !// Stop the program
    STOP
  END IF
  DO j = 1, m
    WRITE(UNIT=olun,FMT='(F4.1,A1,F4.1)',IOSTAT=res) C(i,1), &
    ' ', C(i,2)
    !// Test whether the operation was successful
    IF(res /= 0) THEN
      !// No, an error occurred, print a message to
      !// the terminal window
      PRINT *, "Error in writing file, status: ", res
      !// Exit the loop
      EXIT
    END IF
  END DO
  !// Close the output file
  CLOSE(UNIT=olun)
END PROGRAM f2c_advanced
```

It is important for the efficiency of the program to know how Fortran accesses a matrix. In fact, it does so columnwise.[7] Thus, all the row elements in a column are accessed before the row elements in the next column. Therefore, as a rule of thumb, *always let the first index be the innermost loop* in Fortran.

A.6.7 Functions and Subroutines

It is not a good programming practice to write one long program that includes all the necessary code in one single source file. It makes the program hard to understand, and difficult to maintain. Consequently, it is common to break the program into smaller parts, or subprograms, called into action by the main program. In Fortran, these subprograms are called `FUNCTION`s or `SUBROUTINE`s.

In what follows, the program in the previous section is broken into a main program and a subprogram. The task of the subprogram is simply to do the conversion from Fahrenheit to Celsius, or the other way around. This may be done either by a function or by a subroutine.

Functions
First the `FUNCTION` is used. In the former program, the conversion was either from Fahrenheit to Celsius or from Celsius to Fahrenheit. Two functions are therefore needed, one to convert from Fahrenheit to Celsius, here called `f2c(arg)`, and a second converting from Celsius to Fahrenheit, here called `c2f(arg)`. In this example `f2c(arg)` takes the form:

```
FUNCTION f2c(F) RESULT(C)
  IMPLICIT NONE
!// The input argument which is read only
  REAL(KIND=8), INTENT(IN) :: F
!// The result from the calculations
  REAL(KIND=8) :: C
!// Perform the calculation
  C = (F -32) * 5./9.
!// Return the result
  RETURN
END FUNCTION f2c
```

while `c2f(arg)` takes the form:

```
FUNCTION c2f(C) RESULT(F)
  IMPLICIT NONE
!// The input argument which is read only
  REAL(KIND=8), INTENT(IN) :: C
!// The result from the calculations
  REAL(KIND=8) :: F
```

[7] This is also true for Matlab.

```
!// Perform the calculation
  F = (C * 9. / 5.) + 32
!// Return the result
  RETURN
END FUNCTION c2f
```

In contrast to how mathematical functions are written, Fortran 90–2003 adds the keyword `RESULT(arg)`, where the data type of the argument `(arg)` states what kind of function it is.[8] Here, `INTENT(IN)` is used for the input argument to the functions to prevent accidental overwriting of the argument, since Fortran function and subroutine arguments are always called by reference and not by value. The only thing the main program needs to do, apart from providing the user interface, is to replace the formula for the conversion with a call to the respective functions. Using the Fortran intrinsic functions, they have to be declared as external functions of the correct type. This is taken care of by the code lines `REAL, EXTERNAL :: f2c` and `REAL, EXTERNAL :: c2f`. If omitted, the compiler will flag en error and the compilation will be aborted. The complete main program is as follows:

```
!////////////////////////////////////////////////////////
!//
!// array2.f90
!//
!// A program converting from Fahrenheit to Celsius
!// and vice versa
!//
!////////////////////////////////////////////////////////
PROGRAM array2
  IMPLICIT NONE
  !// Declare everything .....
  !// 1. Arrays for the temperatures
  REAL,DIMENSION(7,2) :: F   ! Fahrenheit
  REAL,DIMENSION(7,2) :: C   ! Celsius
  !// 2. Index variables
  INTEGER :: i, j
  !// 3. Character strings to hold the filenames
  CHARACTER(LEN=80) :: fahrenheitfile
  CHARACTER(LEN=80) :: centigradefile
  !// 3. Constant values for the Logical Unit Number (lun)
  !//    for referencing the files for opening, reading
  !//    and writing
  INTEGER, PARAMETER :: ilun = 10
  INTEGER, PARAMETER :: olun = 11
  !// 4. A status variable to hold the result of file
  !//    operations
  INTEGER :: res
  !// 5. External function(s)
  REAL, EXTERNAL :: f2c
```

[8] In Fortran 77 and older versions, the syntax was `REAL FUNCTION f2c(arg)`.

```
REAL, EXTERNAL :: c2f
!// 6. A character string for a prompt
CHARACTER(LEN=80) :: prompt
CHARACTER :: answer
!// ..... End declarations
!// Assign filenames for temperatures
fahrenheitfile = "fahrenheit.txt"
centigradefile = "centigrade.txt"
!// Ask if we are to convert from Fahrenheit to
!// Celsius or vice versa
prompt = 'Convert from Fahrenheit to Centigrade (F/C)?'
PRINT *, TRIM(prompt)
READ(*,*) answer
!// Check if the answer is F or f for Fahrenheit
IF(answer .EQ. 'F' .OR. answer .EQ. 'f') THEN
!// Yes, open the Fahrenheit input file
  OPEN(UNIT=ilun,FILE=fahrenheitfile,FORM="FORMATTED", &
  IOSTAT=res)
  !// Test whether the operation was successful
  IF(res /= 0) THEN
  !// No, an error occurred, print a message to
  !// the screen
    PRINT *, "Error in opening file, status: ", res
  !// Stop the program
    STOP
  END IF
  !// Loop and read each line of the file
  DO i = 1, 7
    !// Read the current line
    READ(UNIT=ilun,FMT='(F4.1,X,F4.1)',IOSTAT=res) &
    F(i,1), F(i,2)
    !// Was the read successful?
    IF(res /= 0) THEN
      !// No, Test whether we have reached End Of File (EOF)
      IF(res /= -1) THEN
        !// No, an error has occurred, print a message
        PRINT *, "Error in reading file, status: ", res
        !// Close the file
        CLOSE(UNIT=ilun)
        !// Stop the program
        STOP
      END IF
    END IF
  END DO
ELSE
  !// No, open the Celsius input file
  OPEN(UNIT=ilun,FILE=centigradefile,FORM="FORMATTED", &
  IOSTAT=res)
  !// Test whether the operation was successful
  IF(res /= 0) THEN
    !// No, an error occurred, print a message to
    !// the screen
```

```
      PRINT *, "Error in opening file, status: ", res
      !// Stop the program
      STOP
    END IF
    !// Loop and read each line of the file
    DO i = 1, 7
      !// Read the current line
      READ(UNIT=ilun,FMT='(F4.1,X,F4.1)',IOSTAT=res) &
      C(i,1), C(i,2)
      !// Was the read successful?
      IF(res /= 0) THEN
        !// No, Test whether we have reached End Of File (EOF)
        IF(res /= -1) THEN
          !// No, an error has occurred, print a message
          PRINT *, "Error in reading file, status: ", res
          !// Close the file
          CLOSE(UNIT=ilun)
          !// Stop the program
          STOP
        END IF
      END IF
    END DO
  END IF
  !// Close the input file
  CLOSE(UNIT=ilun)
  !// Which way to convert ?
  IF(answer .EQ. 'F' .OR. answer .EQ. 'f') THEN
    !// Loop and convert from Fahrenheit to Celsius
    DO j = 1, 2
      DO i = 1, 7
        C(i,j) = f2c(F(i,j))
      END DO
    END DO
  ELSE
  !// Loop and convert from Celsius to Fahrenheit
  DO j = 1, 2
    DO i = 1, 7
      F(i,j) = c2f(C(i,j))
    END DO
  END DO
  END IF
  !// Which file to write to ?
  IF(answer .EQ. 'F' .OR. answer .EQ. 'f') THEN
    !// Open the Centigrade output file
    OPEN(UNIT=olun,FILE=centigradefile,FORM="FORMATTED", &
    IOSTAT=res)
    !// Test whether the operation was successful
    IF(res /= 0) THEN
      !// No, an error occurred, print message to the screen
      PRINT *, "Error in opening output file, status: ", res
      !// Stop the program
      STOP
    END IF
```

```
      DO i = 1, 7
        WRITE(UNIT=olun,FMT='(F4.1,A1,F4.1)',IOSTAT=res) &
        C(i,1), ' ', C(i,2)
        !// Test whether the operation was successful
        IF(res /= 0) THEN
          !// No, an error occurred, print message to the screen
          PRINT *, "Error in writing file, status: ", res
          !// Exit the loop
          EXIT
        END IF
      END DO
    ELSE
      !// Open the Fahrenheit output file
      OPEN(UNIT=olun,FILE=fahrenheitfile,FORM="FORMATTED", &
      IOSTAT=res)
      !// Test whether the operation was successful
      IF(res /= 0) THEN
        !// No, an error occurred, print message to the screen
        PRINT *, "Error in opening output file, status: ", res
        !// Stop the program
        STOP
      END IF
      DO i = 1, 7
        WRITE(UNIT=olun,FMT='(F4.1,A1,F4.1)',IOSTAT=res) &
        F(i,1), ' ', F(i,2)
        !// Test whether the operation was successful
        IF(res /= 0) THEN
          !// No, an error occurred, print message to the screen
          PRINT *, "Error in writing file, status: ", res
          !// Exit the loop
          EXIT
        END IF
      END DO
    END IF
    !// Close the output file
    CLOSE(UNIT=olun)
  END PROGRAM array2
```

In contrast to functions, a subroutine does not return a value and is the same as a void function in other languages. The functions f2c() and c2f() may be replaced by corresponding subroutines, for instance, which can look like this:

```
!////////////////////////////////////////////////////////
!//
!// SUBROUTINE f2c(F,C)
!//
!// Called from program array2.f90
!//
!////////////////////////////////////////////////////////
SUBROUTINE f2c(F,C)
  IMPLICIT NONE
```

```
    !// The input argument which is read only
    REAL(KIND=8), INTENT(IN) :: F
    !// The result of the conversion which is
    !// write only
    REAL(KIND=8), INTENT(OUT) :: C
    !// Perform the conversion
    C = (F -32) * 5./9.
    !// Return the result through the second argument
    RETURN
  END SUBROUTINE f2c
```

and

```
!////////////////////////////////////////////////////////
!//
!// SUBROUTINE c2f(C,F)
!//
!// Called from program array2.f90
!//
!////////////////////////////////////////////////////////
SUBROUTINE c2f(C,F)
  IMPLICIT NONE
  !// The input argument which is read only
  REAL(KIND=8), INTENT(IN) :: C
  !// The result of the conversion which is
  REAL(KIND=8), INTENT(OUT) :: F
  !// Perform the conversion
  F = (C * 9. / 5.) + 32
  !// Return the result through the second argument
  RETURN
END SUBROUTINE c2f
```

The call from the main program is like this:

```
    !// Loop and perform the conversion using
    !// nested loops
    DO j = 1, 2
      DO i = 1, 7
      !// Call the subroutine with the current element
      !// of the Fahrenheit array as the first argument to
      !// the subroutine and the current element of the
      !// centigrade array as the second argument.
        CALL f2c(Fahrenheit(i,j),centigrade(i,j))
      END DO
    END DO
```

and

```
    !// Loop and perform the conversion using
    !// nested loops
```

```
DO j = 1, 2
  DO i = 1, 7
  !// Call the subroutine with the current element
  !// of the Fahrenheit array as the first argument to
  !// the subroutine and the current element of the
  !// centigrade array as the second argument.
    CALL c2f(centigrade(i,j).Fahrenheit(i,j))
  END DO
END DO
```

Only non-intrinsic functions can be declared as external, subroutines cannot. Note also the use of `INTENT(OUT)`, which means that one can only give value to the argument (write only). Trying to read the value from the argument would flag a compilation error, just as it would if we were trying to give the argument a value when it had the attribute `INTENT(IN)` (read only).

A.7 Exercises

1. Use the code in Sect. A.6, fill in what is missing, and save the source code in a file. Compile the code and run it to check that, in a non-leap year, daynumber for September 16 is indeed 259.
2. Given the radius of a circle, write a program to calculate the length of the circumference of the circle. Compile and run the program to check that the result is correct.
3. Given the radius of a circle, write a program to calculate the area inside the circle. Compile and run the program to check that the result is correct.
4. Given the radius of a sphere, write a program to calculate the volume of the sphere. Compile and run the program to check that the result is correct.