

# 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”,<sup>1</sup> which precisely defines what represents a valid program. This helps the compiler to unambiguously interpret the programmer’s source code,<sup>2</sup> 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):

---

<sup>1</sup> For example, *extended Backus-Naur form* (EBNF).

<sup>2</sup> EBNF is also useful for defining consistent data formats and even simple *domain-specific languages* (DSLs).

```

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.<sup>3</sup> 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*,<sup>4</sup> *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

<sup>3</sup> 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.

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

letters, digits or underscores `_`). The length of the identifiers should not exceed 63 characters (Fortran 2003 or newer).<sup>5</sup>

**Comments:** Commenting the nontrivial aspects of your code is highly recommended.<sup>6</sup> In Fortran, this is achieved by typing an exclamation mark (`!`), either on a line of its own, or within another line which also contains program code. In either case, an `!` will cause the compiler/preprocessor to ignore the rest of the line.<sup>7</sup>

**Multi-line statements:** Unlike languages from the C-family, in Fortran the semicolon `;` for marking the end of a statement is optional (although it is still used sometimes, to pack several short statements on the same line). By default, the end of the line is also considered to be the end of the statement. A line of code in Fortran should be at most 132 characters long. If a statement is so long that this is not sufficient (for example, a long formula for evaluating derivatives in finite-difference numerical schemes), we can choose to continue it on the following line(s), by inserting an ampersand `&` at the end of each line that is continued. Since Fortran 2003, up to 255<sup>8</sup> continuation lines are allowed for any statement.

It can happen (although it should be avoided when possible) that the line break in a multi-line statement occurs at the middle of a token. In that case, using a single `&` will probably not give the expected result. This can be overcome by typing another `&` as the first character on the continued line, which contains the remainder of the divided token.

The two possible uses of continuation lines are shown in the example below:

```

1  program continuation_lines
2      implicit none
3      integer :: seconds_in_a_day = 0
4
5      ! Normal continuation-lines
6      seconds_in_a_day = &
7          24*60*60 ! 86400
8
9      print*, seconds_in_a_day
10
11     ! Continuation-lines with a split integer-literal token
12     seconds_in_a_day = 2&
13         &4*60*60 ! still 86400. In this case, splitting the '24'
14             ! is unwise, because it makes code unreadable.
15             ! However, for long character strings this can be
16             ! useful (see below).
17     print*, seconds_in_a_day
18
19     ! Continuation-lines with a split string token.
20     print*, "This is a really long string, that normal &
21         &ally would not fit on a single line."
22 end program

```

**Listing 2.2** `src/Chapter2/continuation_lines.f90`

<sup>5</sup> A maximum of 31 characters were allowed in Fortran 95.

<sup>6</sup> A good guideline is to make the code indicate clearly **what** is being done (through choice of meaningful variable and function names), and then to use the comments to describe the motivations (**why** it has been done like that, and what other problem-specific aspects are relevant).

<sup>7</sup> Exceptions to this rule are compiler directives (“pragmas”), which are specially-formatted comments that communicate additional information to the compiler; examples will be shown in Sect. 5.3, when we will discuss how to specify, using the *Open MultiProcessing* (OpenMP) extensions, which portions of the code should be attempted to be run in parallel.

<sup>8</sup> The previous limit (according to the Fortran 95 standard) was of up to 39 continuation lines.

**Spaces and indentation:** Whitespace can be freely used to separate program tokens, without changing the meaning of the program. For example, as far as the compiler is concerned, *line 3* in the previous listing could also have been written as:

```
integer ::seconds_in_a_day= 0
```

Therefore, this is a subjective choice, which can be used to our advantage, to improve readability of the code. For example, it is considered good practice to indent<sup>9</sup> program-flow constructs (loops, conditionals, etc.), as will be shown later.

**Combining statements in one line:** As previously mentioned, we normally have one statement per line of code. However, it is also allowed to combine instructions, as long as they are separated by semicolons `;`. A common example is for swapping two variables (`a` and `b`) using a temporary (`temp`):

```
temp=a; a=b; b=temp ! semicolon not mandatory at end of line
```

## 2.3 Scalar Values and Constants

As other programming languages, Fortran allows us to define named entities, for representing quantities of interest from the problem domain (speed, temperature, concentration of a tracer, etc.). Each entity belongs to a type, which specifies a *set of possible values*, a *scheme for encoding those values*, and *allowed operations*. Fortran is *statically-typed*, which means the type of a variable is fixed at compile-time.<sup>10</sup> This apparent drawback actually helps in practice, because many errors can be caught earlier; it also helps the compiler to apply certain code-optimization techniques (because the number of bits needed for each variable is known well before any operation is applied to the variable).

Standard-compliant compilers should provide at least five built-in types.<sup>11</sup> Of these, three are *numeric* (`integer`, `real` and `complex`), and two *non-numeric* (`character` and `logical`).

<sup>9</sup> Note that some text editors feature automatic indentation, which makes this easier.

<sup>10</sup> Other languages, such as *Matrix Laboratory*<sup>®</sup> (MATLAB) or *The R Project for Statistical Computing* (R), support *dynamic typing*, so the type of a variable can change during the execution of the program.

<sup>11</sup> It is also possible to define custom types, enabling data-encapsulation techniques similar to C++ (this will be discussed in Sect. 3.3.2).

All of these types can be used to declare *named constants* or *variables*:

```
! Declaring normal variables
! -- numeric --
integer :: length = 10
real    :: x = 3.14
complex :: z = (-1.0, 3.2)
! -- non-numeric --
character :: keyPressed = 'a'
logical   :: condition = .false. ! (either '.true.' OR '.false.')
```

```
! Declaring named constants
! -- numeric --
integer, parameter :: INT_CONST = 30
real, parameter    :: REAL_CONST = 1.E2 ! (scientific notation)
complex, parameter :: I = (0.0, 1.0)
! -- non-numeric --
character, parameter :: B_CHAR = 'b'
logical, parameter   :: IS_TRUE = .true.
```

## NOTES

- **Position of declarations in (sub)programs:** All declarations for constants and variables need to be included at the beginning of the (sub)program, *before* any executable statement. However, as of Fortran 2008 it is possible to overcome this limitation, by surrounding variable declarations with a `block`-`end block` construct, as follows:

```
! variable declaration(s)
integer :: length

! executable statements (normally, not possible to specify additional
! variable declarations after the first such executable statement)
length = 10

block
! block-construct (Fortran 2008+) enables us to overcome that
! limitation
real :: x
end block
```

- **The `::` separator:** In the examples below, we declare variables both with and without this separator. In general, `::` is optional, except when *the variable is also initialized* or *variable attributes are specified*. A simple rule of thumb is to always use this separator, which works in all cases.
- **Constants:** Any value can be declared as a constant, by appending the `parameter`-attribute after the name of the type. *This should be used generously whenever values are known to be constant, to avoid accidental overwriting.* Other type attributes will also be discussed.

### 2.3.1 Declarations for Scalars of Numeric Types

Below, we present some examples of defining variables and constants of numeric types. For each type, we first demonstrate how to define a variable. A definition only reserves space for the variable in memory, but the value stored in its corresponding bits is actually undefined, so it is highly recommended to follow each declaration

with an initialization. These two steps can be merged into a one-liner (see examples below). Finally, we also show how to define constants of each type.

**integer type:** valid values of this type are, e.g.  $-42$ ,  $24$ ,  $123$ . In general, any integer is accepted, as long as it resides within a certain range. The length of the range is determined by the `kind` parameter (if that is explicitly specified), or by the machine architecture (32 or 64 bit) and compiler (if no `kind` is specified, as in our present examples). Example declarations:

```
integer i           ! plain declaration...
i = 10             ! ...with corresponding initialization
                  ! (would be in the executable section of
                  ! the (sub)program)
integer            :: j = 20 ! declaration with initialization
integer, parameter :: K = 30 ! constant (initialization mandatory)
```

Note that, unlike other programming languages, Fortran integer-variables are always signed (i.e. they can be both positive and negative).

**real type:** valid values of this type are, e.g.  $2.78$ ,  $99.$ ,  $1.27e2$  (exponential notation)<sup>12</sup> or  $.123$ . Similar to integers, the number of digits after the decimal point (precision) and range of the exponent are system- and kind-dependent. Example declarations:

```
real x              ! simple declaration
real               :: y = 1.23 ! declaration with initialization
real, parameter   :: Z = 1.e2 ! constant (scientific notation)
```

**complex type:** complex numbers are often needed in scientific and engineering applications, thus Fortran supports them natively. They can be specified as a pair of integer or real values (however, even if both components are specified as integers, they will be stored as a pair of reals, of default kind). Example declarations:

```
complex c1 ! simple declaration
complex :: c2 = (1.0, 7.19e23) ! declaration with initialization
complex, parameter :: C3 = (2., 3.37) ! constant
```

### 2.3.2 Representation of Numbers and Limitations of Computer Arithmetic

While internally all data is stored by computers as a sequence of bits (zeroes and ones), the concept of types (also known as “data types”) binds the byte sequences to specific interpretations and manipulation rules. For example, addition of **integer** versus that of **real**-numbers is very different at the bit-level. The number of bits used for a value of each type is particularly important: the more bits are used, the

<sup>12</sup>  $1.27e2 \equiv 1.27 \times 10^2$ .

more numbers become representable. For `integer`s, this is exploited to increase the bounds of the representable interval; for `real`s, part of the extra bits can be used to increase the *precision* (i.e. roughly the number of digits after the decimal point, after we translate the number back to the decimal representation). As a rule of thumb, computations become more expensive when more bits are used.<sup>13</sup> However, numerical algorithms also vary with respect to the precision they need to function correctly. To balance these factors, most computer systems support several sub-types for `integer` and `real` values.

Modern Fortran has a very convenient mechanism for specifying the numerical requirements of a program in a portable way, without forcing developers (or, worse, users) to study each CPU in-depth. We discuss this feature in Sect. 2.3.4.

It is important that programmers keep in mind the limitations of the internal representations, since these are an endless source of bugs. A tutorial on these issues is outside the scope of our text (a very readable introduction to these issues and their implications is Overton [11]). For example, some of the facts to keep in mind for the `integer` and `real` types are:

- `integer`: Unlike C, Fortran always stores integer-values with a sign.<sup>14</sup> All `integer` types represent *exactly* all numbers within a sub-interval of the mathematical set of integer numbers. However, different kinds of integers (using different number of bits) will have different lengths for the representable interval. This is important when our programs use conversions from one kind of integer to another. Also, operations involving two integers may produce a result which is not representable inside the type (a situation known as *integer overflow*). Sometimes, compilers may have options which can detect such errors when a program is tested – for example, `gfortran` can achieve this when the `-ftrapv` flag is used.<sup>15</sup>
- `real`: Most computer systems nowadays support the `IEEE 754` standard. This specifies a set of rules for representing fractional numbers inside the computer, along with bounds on the errors they introduce. This representation is also known as “floating-point”, since it was inspired by the floating-point representation of large numbers, used in science and engineering calculations. While integer-arithmetic is exact (as long as both the arguments and the result are representable), this is not the case for floating-point representations: since any interval along the real axis contains an infinite set of numbers, it is impossible to store most

---

<sup>13</sup> This is not a “hard” rule, however, because many factors enter the performance equation—e.g. specialized hardware units within the *central processing unit* (CPU), the memory hierarchy, vectorization, etc.

<sup>14</sup> An intuitive approach would be to reserve one bit for the sign, and use the rest for the modulus. However, to reduce hardware complexity most systems use another convention (“two’s complement”).

<sup>15</sup> Note that enabling such options will most probably make the program slower too, so they are not meant for “production” runs.

numbers using a bit-field of finite size. This causes most nontrivial calculations with *real*-values in Fortran to be *approximate* (so there is always some “noise” in the numerical results).

To complicate matters more, note that *many numbers which are exactly representable in the familiar decimal floating-point notation cannot be represented exactly when translated to one of the binary floating-point formats*. A common example is the number 0.1, which on our system becomes 0.100000001490116 when translated to 32 *bit* floating-point and back to decimal, and 0.1000000000000000005551115123125783 when 64 *bit* floating-point is used. This can lead to subtle bugs—for example, when two variables which were both assigned the value 0.1 are compared for equality, the result may be *false* if the two variables are of different floating-point type. In this case, the compiler will promote the lower-precision value to the higher-precision type (so that it can perform the comparison). However, this cannot bring back the bits that were discarded when 0.1 was converted to fit inside the lower-precision type. For this reason, it is often a good idea to avoid such comparisons as long as possible, or to include some tolerances when this operation is necessary nonetheless. For more information on floating-point arithmetic and advice for good practices, the reader can also consult Goldberg [4], as well as Overton [11].

### 2.3.3 Working with Scalars of Numeric Types

The three numeric types share some characteristics, so it makes sense to discuss their usage simultaneously, highlighting any exceptions. This is the purpose of this section.

#### 2.3.3.1 Constructing Expressions

Scalars of numeric type (*operands*), together with *operators*, can be combined to form *scalar expressions*. Fortran supports the usual *binary* arithmetic operators: `**` (exponentiation), `*` (multiplication), `/` (division), `+` (addition), and `-` (subtraction). The last two may also be used as *unary* operators (for example, to negate a value). Complex expressions can be built, with more than one operator. For evaluation, these are divided into sub-expressions, which are executed in left-to-right order, with some complications due to the precedence rules (for the details, consult, for example, Metcalf et al. [10]). Parentheses can be used to override the precedence rules, which may make code readable in some cases:

```
real :: x=13, y=17, z=0
z = x*y+x/y ! this expression (using precedence rules)
z = (x*y) + (x/y) ! is equivalent to this one (using parentheses)
```



### 2.3.3.2 Mixed-Mode Expressions

The generality of numeric types in Fortran mirrors their mathematical counterparts:  $\mathbb{Z} \subset \mathbb{R} \subset \mathbb{C}$ . When operands in a numeric expression do not have the same type and kind, Fortran usually converts the less precise/general operand, to conform to the more precise/general of the operands. A notable exception to this rule is when raising a real to an integer-power, in which case the integer is not converted to real (which is good, since raising to an integer power is more accurate and faster than raising to a corresponding real power). Another, less fortunate exception is when one of the operands is a literal constant, which can lead to loss of precision (therefore it is recommended to ensure the kind of the constant is specified—how to do this will be shown in Sect. 2.3.4).

### 2.3.3.3 Using Scalar Expressions

Standalone numerical expressions do not make much sense (hence the language does not allow them): what we actually want is to *assign* the result of the expressions to variables (with the `=` assignment operator<sup>16</sup>), or to *pass* the result to some function (e.g. to display it). This is another point where loss of precision can occur, if the expression is of a stronger type/kind than the variable to which it is assigned:

```
integer :: i = 0
real :: m = 3.14, n = 2.0
i = m / n      ! i will become 1, NOT 1.57 (rounding towards 0)
m = -m         ! negate m with unary operator
i = m / n      ! i will become -1 (rounding also towards 0)
print *, m / n ! expression passed to 'print'-function
```

### 2.3.3.4 Convenient Notation for Sub-components of complex

For applications that need to work with data of complex type, note that it is possible (since Fortran 2008) to conveniently refer to the real and imaginary components:

```
complex :: z1(1.0, 2.0)
z1%im = 3.0 ! modify the imaginary part
print *, "real part of z1 = ", z1%re
```

## 2.3.4 The *kind* type-parameter

Most of the numerical algorithms encountered in ESS need some assumptions regarding properties of the types used to represent the quantities they manipulate. For example, if integers are used to represent simulation time in seconds, we need to ensure the type can support the maximum number of seconds the model will be run for. The

<sup>16</sup> Not to be confused with an equivalence in the mathematical sense. In Fortran, that is represented by the `==` operator, which we discuss shortly, in relation to the `logical` type.

demands are more complex for reals, which are always stored with finite precision: since each result needs to be truncated to fit the representation, numerical “noise” is ever-present, and needs to be constrained.

One way to improve<sup>17</sup> the situation is to increase the accuracy of the representation, by using more bits to represent each value. In older versions of Fortran, the `double precision` type (real-variant) was introduced exactly for this. The problem, however, lies in the fact that the actual number of bits is still system- and compiler-dependent: switching hardware vendors and compilers is normal, and surprises due to improper number representation (which can often go unnoticed) are better avoided when possible.

The concept of `kind` is the modern Fortran response to this problem, and it deprecates the `double precision` type. `kind` acts as a parameter to the type, allowing the programmer to select a specific type variant from the multitude that may be supported by the platform.<sup>18</sup> Even better, the programmer need not be concerned with the lower-level details, since two special intrinsic functions (discussed shortly) allow querying for the most economic types that meet some natural requirements.

We only discuss `kind` for numeric types, although the concept also applies to non-numeric types (for storing characters of non-European languages and for efficiently packing arrays of `logical` type—for details consult, e.g. Metcalf et al. [10]).

Kinds are indexed with positive integer values. If we know these indices for selecting numbers with the desired properties on the current platform, they can be used to parameterize types, as in:

```
integer( kind=4) :: i
real( kind=16) :: x
```

However, this feature alone does not solve the portability problem, since the index values themselves are not standardized. The intended usage, instead, is through two intrinsic functions which return the appropriate `kind`-values, given some constraints requested by the developer:

1. `selected_int_kind(requestedExponentRange)`, where `requestedExponentRange` is an integer, returns the appropriate `kind`-parameter for representing integer numbers in the range:

$$-10^{\text{requestedExponentRange}} < \textit{number} < 10^{\text{requestedExponentRange}}$$

For example,<sup>19</sup>

```
integer, parameter :: LARGE_INT = selected_int_kind(18)
integer(kind=LARGE_INT) :: t = -123456_LARGE_INT
```

<sup>17</sup> This is not to be seen as a “silver bullet”, since numerical noise will still corrupt the results, if the algorithm is inherently unstable.

<sup>18</sup> Compilers are required to provide a default `kind` for each of the 5 intrinsic types, but they may (and most do) support additional kinds.

<sup>19</sup> In practice, it is more convenient to use shorter denominators for the `kind`-parameters.

will guarantee that the compiler selects a suitable type of integer to fit values of  $t$  in the interval  $(-10^{18}, 10^{18})$ .

2. `selected_real_kind(requestedPrecision, requestedExponentRange)`, where both arguments are integers, returns the appropriate kind-parameter for representing numbers with a *decimal exponent range* of at least `requestedExponentRange`, and a *decimal precision* of at least `requestedPrecision`<sup>20</sup> after the decimal point.

Example:

```
integer, parameter :: MY_REAL = selected_real_kind(18,200)
real(kind=MY_REAL) :: x = 1.7_MY_REAL
```

To obtain what is commonly denoted as single-, double-, and quadruple-precision, the following parameters can be used:

```
integer, parameter :: R_SP = selected_real_kind(6,37)
integer, parameter :: R_DP = selected_real_kind(15,307)
integer, parameter :: R_QP = selected_real_kind(33,4931)
```

Note that, since the exact data type needs to be revealed to the compiler, results of the kind-inquiries need to be stored into *constants* (which are initialized at compile-time).

By increasing the values of the `requestedExponentRange` and/or `requestedPrecision` parameters, it is easily possible to ask for numbers beyond the limits of the platform (you will get the chance to test this in Exercise 7). In such situations, the inquiry functions will return a negative number. This fits with the way kind type-parameters are used, since trying to specify a negative kind value will cause the compilation to fail:

```
integer, parameter :: NONSENSE_KIND = -1
integer(kind=NONSENSE_KIND) :: s ! will fail to compile

integer, parameter :: UNREASONABLE_RANGE = selected_int_kind(30000)
! will also fail to compile (at least, in 2013), because a too ambitious range
! of values was requested, causing the intrinsic function to
! return a negative number
integer(kind=UNREASONABLE_RANGE) :: u
```

In closing of our discussion on `kind`, we have to admit that inferring the type-parameters in each (sub)program, while viable for simple examples, can become tedious and, worse, leads to much duplication of code. An elegant solution to this problem is to package this logic inside a `module`, which is then included in (sub)programs.<sup>21</sup> We defer the discussion of this mechanism to Sect. 3.2.7, after covering the concept of modules.

<sup>20</sup> The situation is more complex for this type, because some values which are exactly representable using the *decimal* floating-point notation can only be approximated in the *binary* floating-point notation.

<sup>21</sup> We encountered this mechanism in the Fortran distribution of the popular *Numerical Recipes* book, see Press et al. [12].

### 2.3.5 Some Numeric Intrinsic Functions

As a language designed for science and engineering applications, Fortran supports a large suite of mathematical functions, which complement the operators. Also, including these as part of the core language allows vendor-specific implementations to take advantage of special hardware support for some costly functions.

Among the most frequently-used numeric intrinsic functions, we mention:

- *type conversion*: `real(x [, kind])`, `int(x [, kind])`
- *trigonometric functions* (operating with radians): `sin(x)`, `cos(x)`, `tan(x)`; also, inverse ( `asin(x)`, `acos(x)`, `atan(x)` ), and hyperbolic functions ( `sinh(x)`, `cosh(x)`, `tanh(x)` )
- *usual mathematics*: `abs(x)` (absolute value), `exp(x)`, `log(x)` (natural logarithm), `sqrt(x)`, `mod(x [,n])` (remainder modulo-*n*)

This list is by no means comprehensive (see Metcalf et al. [10] for an exhaustive version). The `kind` of the result is usually the same as that of the first parameter, unless the function accepts a `kind`-parameter, which is present.

In Sect. 2.7, we discuss some more intrinsic functions, useful for more advanced tasks.

### 2.3.6 Scalars of Non-numeric Types

`logical type`: allows variables to take only two values: `.true.` or `.false.` (dots are mandatory). They can be declared similarly to the other types:

```
logical activated      ! plain declaration...
activated = .true.    ! ...with corresponding initialization
logical :: conditionSatisfied = .false. ! declaration with init
logical, parameter :: ON = .true. ! constant (init mandatory)
```

**logical expressions**: As for numeric types, `logical` values can be used, together with specific operators (unary: `.not.`; binary: `.and.`, `.or.`, `.eqv.` (equality) and `.neqv.`), to construct expressions, as in (using the previous declarations):

```
.not. conditionSatisfied ! .eqv. .true.
conditionSatisfied .and. ON ! .eqv. .false.
```

It is important to know that `logical` expressions can also be constructed out of numeric arguments, using the arithmetic *comparison operators*: `==` (equal), `/=` (not equal), `>` (greater), `>=` (greater-or-equal), `<` (smaller), and `<=` (smaller-or-equal). Such `logical` expressions are used in flow-control statements (e.g. `if`), discussed in Sect. 2.5.

**character type:** variables and constants of this type are used for storing text characters, similar to other programming languages. In Fortran, characters and character strings are marked by *a pair of single quotes* (as in ‘some text’), or *a pair of double quotes* (as in “some more text”). These can be used interchangeably, both for single- and multi-character values.

A text character is said to belong to a character set. A very influential such character set is ASCII, which can be used to represent English-language text. Ours being an English text, we devote more space to this character set.

Many modern Fortran implementations currently use ASCII by default. For example, this is the case on our test system (64bit Linux, gfortran v4.8.2), when we declare variables such as:

```
character char1 ! plain declaration (to be initialized later)
character :: char2 = 'a' ! declaration with immediate initialization
```

We discussed earlier (in the context of numeric types) the concept of **type-parameters**. The character type actually accepts two such parameters: **len** (for controlling the length of the string) and **kind** (for selecting the character set).

Let us focus on the first parameter (**len**) for now. It exists because most of the times developers want to store *sequences of characters (strings)*. If (as in the previous listing) **len** is not explicitly mentioned, it implicitly has the value fixed to “1” (reserving space for just one ASCII-character). To store wider strings, we can declare a sufficiently-large value for **len**, e.g.:

```
character(len=100) myName ! fixed-size string
```

However, this method is not so convenient in practice.

For the case where the length of the string can be determined during compilation (i.e. it will not change when our program will be executed), we can use *assumed-length* strings. This is particularly useful for declaring constant strings, sparing the developer from counting characters (for the **len** parameter):

```
1 program assumed_length_strng_constant
2   implicit none
3
4   character(len=*), parameter :: FILENAME = & ! character constant
5     'really_long_file_name_for_which_&
6     &we_do_not_want_to_count_characters.out'
7
8   print*, 'string length:', len(FILENAME)
9 end program assumed_length_strng_constant
```

**Listing 2.3** src/Chapter2/assumed\_length\_strng\_constant.f90

Note the type-parameter **len=\*** (line 4), which causes the string to have what is known as *assumed-length*.

Another common scenario is when the strings to operate on are not constant, with their lengths only becoming known during the execution of the program. This is the case, for example, if we want to make the previous listing more flexible, by asking

the user to provide a filename.<sup>22</sup> For such a situation, we can use *deferred-length* strings, which are marked by the type-parameter `len=:`, in conjunction with the specifier `allocatable`. For example:

```

1 program deferred_length_strng
2   implicit none
3
4   character(len=256) :: buffer ! fixed-length buffer
5   character(len=:), allocatable :: filename ! deferred-length string
6
7   print*, 'Please enter filename (less than 256 characters):'
8   read*, buffer ! place user-input into fixed buffer
9
10  filename = & ! copy from buffer to dynamic-size string
11         trim(adjustl(buffer)) ! 'trim' and 'adjustl' explained later
12
13  print*, filename ! some feedback...
14 end program deferred_length_strng

```

**Listing 2.4** `src/Chapter2/deferred_length_strng.f90`

It is not possible to place a value in `filename` directly from the `read`-statement (line 8). Therefore, we declare an extra buffer to hold the input data (line 4). The actual *deferred-length* variable is declared at line 5. On line 7 we announce to the user that a string (i.e. characters surrounded by single or double quotes!) is expected, and on line 8 we read the input into the buffer. At line 10 we finally get to use the deferred-length variable. Ignoring the intrinsic functions for now, the net effect is that a string will be assigned to `filename`. Note that the system automatically reserves memory internally, so that our variable `filename` is large enough. Later, we will also discuss how to *explicitly* request such memory, in the context of dynamic arrays (Sect. 2.6.8).

**character operators and intrinsic functions:** For the character type it is useful to know about the operator `//`, which concatenates two strings. Expressions formed with this operator are strings with length equal to the sum of the lengths of the strings to be concatenated. We usually want to assign the evaluated expressions to other string variables, in which case truncation or whitespace-padding (on the right) can occur, depending on the length of the expression and of the string variable we assign to. These situations are illustrated in the following example<sup>23</sup>:

```

program character_type_examples
  implicit none

  ! given two source string-variables:
  character(len=4) :: firstName = "John"
  character(len=7) :: secondName = "Johnson"
  ! and 3 target variables (of different lengths):
  character(len=13) :: exactFit
  character(len=10) :: shorter
  character(len=40) :: wider =&
    "Some phrase to initialize this variable."

```

<sup>22</sup> This approach is more convenient, in the sense that the user does not have to re-compile the program every time the filename changes. For real-world software, we prefer to minimize interaction with users, and allow specification of filenames (e.g. model input) at the invocation command line instead (Sect. 5.5.1), which facilitates unattended runs.

<sup>23</sup> If you try this example, you may notice that an additional space is printed at the beginning of every line. This is the default behavior, related to some legacy output devices. We will discuss how to avoid this in Sect. 2.4.

```
! below, we concatenate 'firstName' and 'secondName',
! assigning the result to strings of different sizes.
! note: '/'-characters serve as markers, to highlight
! the spaces in the actual output.

! expression fits exactly into 'exactFit'
exactFit = firstName //"", "// secondName
print*, "|", exactFit, "|"

! expression does not fit into 'shorter', so some
! characters at the end are truncated
shorter = firstName //"", "// secondName
print*, "|", shorter, "|"

! expression takes less space than available in
! 'wider', so whitespace is added as padding on
! the right (previous content discarded)
wider = firstName //"", "// secondName
print*, "|", wider, "|"
end program character_type_examples
```

**Listing 2.5** src/Chapter2/character\_type\_examples.f90

**Table 2.1** Some intrinsic functions for character(-strings)

| Function name                                     | Result/Effect   |
|---|---|
| lge(string1, string2)<br>(similar: lgt, lle, llt) | .true. if string1 follows after or is equal to string2<br>(lexical comparison, based on ASCII collating sequence) |
| len(string)                                       | length of string  |
| trim(string)                                      | string, excluding trailing padding-whitespace   |
| len_trim(string)                                  | length of string, excluding trailing padding-whitespace   |
| adjustr(string)<br>(similar: adjustl)             | right-justify string  |

In ESS models, characters and strings are often secondary to the core numerics. They are, however, useful for manipulating model-related *metadata*. To cater for such needs, Fortran provides several intrinsic functions that take strings arguments (see Table 2.1 for a basic selection, or Metcalf et al. [10] for detailed information).

At the end of Sect. 2.5, after introducing more language elements, we use some of these intrinsic functions, to solve a common pattern in ESS (creation of unique filenames for transient-phenomena model output, based on the index of the time step).

2.4 Input/Output (I/O)

The I/O system is an essential part of any programming language, as it defines ways in which our programs can interact with other programs and with users.

For example, models in ESS typically read files (input) for setting-up the geometry of the problem and/or for loading initial conditions. Then, as the prognostic variables are calculated for the subsequent time step, the new model state is regularly written to other files (output). Frequently, the input files are created in an automatic fashion,

using other programs; likewise, the output of the model may be passed to post-processing/visualization tools, for analysis.<sup>24</sup>

External files are not the only medium for performing I/O; other interfaces include the usual interaction with the user via the terminal, or communication with the *operating system* (OS) (which allows the program to become aware of command line arguments passed to it, and of environment variables—see Sect. 5.5 for some examples). It is also possible to construct *graphical user-interface* (GUI)-based I/O applications, using third-party libraries.<sup>25</sup> but, in ESS, models providing such features<sup>26</sup> are still the exceptions rather than the rule.<sup>27</sup>

We already used some simple I/O-constructs, in the code samples presented so far. In this section, we provide the background for these constructs, and also discuss other aspects of *formatted*<sup>28</sup> I/O (such as controlling the I/O commands, or working with files). Finally, we provide a hierarchical overview of the I/O facilities used in ESS.

### NOTE

*A distinguishing characteristic of Fortran is that, by default, its I/O subsystem is record-based (unlike languages like C or C++, which treat I/O as a stream of bytes<sup>a</sup>).*

<sup>a</sup>This difference can cause problems while exchanging files across languages. Such problems can be avoided by using portable formats like *NETwork Common Data Format* (netCDF) (Sect. 5.2.2) or, when the file format cannot be changed, by using the new stream I/O capabilities of Fortran 2003 (see Metcalf et al. [10]).

## 2.4.1 List-Directed Formatted I/O to Screen/from Keyboard

The simplest form of I/O in Fortran, which we have used so far, enables communication with the program from the terminal where the program was launched. Here, data needs to be converted between the internal representation of the format, and the

<sup>24</sup> The complete network of tasks for obtaining the final data products can become quite complex. In such cases, it often pays off to automatize the entire process, using shell scripts (see Sect. 5.6.1 for a brief overview of the options available, and some suggestions for further reading).

<sup>25</sup> See, for example, *Java Application Programming Interface* (JAPI) for an open-source solution; a commercial alternative is *Winteracter*.

<sup>26</sup> A model which provides a GUI is the *Planet Simulator* (see Fraedrich et al. [3], Kirk et al. [7]).

<sup>27</sup> A lack of graphical interfaces does not imply obsolete software practices: textual, command line interfaces can be readily used to automate complete workflows. This paradigm is suitable for ESS models, which usually need a long time to run. However, GUI-based systems are often suitable for steering operations which complete very fast, such as low-resolution models or tools in exploratory data analysis.

<sup>28</sup> In Fortran, *formatted* I/O means ASCII-text form; conversely, *un-formatted* I/O means binary form. We do not cover binary I/O in this text, even if it is more space-efficient, due to possible portability issues (we highlight an alternative form of efficient I/O in Sect. 5.2.2).



character strings recognized by the terminal. The programmer would often want to control this conversion process, to achieve the desired formatting.<sup>29</sup> However, for testing purposes, the `read*` and `print*` forms can be used, known as *list-directed I/O*. These are demonstrated in the following program, which expects the user to enter a name and date of birth (year, month, day), and returns the corresponding day of the week:

```

program birthday_day_of_week
  implicit none
  character(len=20) :: name
  integer :: birthDate(3), year, month, day, dayOfWeek
  integer, dimension(12) :: t = &
    [ 0, 3, 2, 5, 0, 3, 5, 1, 4, 6, 2, 4 ]

  print*, "Enter name (inside apostrophes/quotes):"
  read*, name
  print*, "Now, enter your birth date (year, month, day):"
  read*, birthDate

  year = birthDate(1); month = birthDate(2); day = birthDate(3)

  if( month < 3 ) then
    year = year - 1
  end if

  ! Formula of Tomohiko Sakamoto (1993)
  ! Interpretation of result: Sunday = 0, Monday = 1, ...
  dayOfWeek = &
    mod( (year + year/4 - year/100 + year/400 + t(month) + day), 7)

  print*, name, " was born on a "
  select case(dayOfWeek)
  case(0)
    print*, "Sunday"
  case(1)
    print*, "Monday"
  case(2)
    print*, "Tuesday"
  case(3)
    print*, "Wednesday"
  case(4)
    print*, "Thursday"
  case(5)
    print*, "Friday"
  case(6)
    print*, "Saturday"
  end select
end program birthday_day_of_week

```

**Listing 2.6** `src/Chapter2/birthday_day_of_week.f90`

The part of the I/O statements following the comma is called an *I/O list*. For input, this needs to consist of variables (also arrays), while for output any expression can be used.

Previously, we mentioned the record metaphor used by Fortran; this needs to be considered while feeding input at the terminal for a `read`-statement: each statement expects its input to span (at least one) distinct line (=record), so before any subsequent `read*`-statement is executed, the file “cursor” would be advanced to the next record, making it impossible to enter on a single line input for adjacent `read*`-statements.

<sup>29</sup> This is discussed later, in Sect. 2.4.2; the process is controlled via an *edit descriptor*, which is embedded in a *format specification*.

Thus, it is perfectly acceptable to write something like:

```

program read_3variables_on_a_line
  implicit none
  character(len=100) :: station_name ! fixed-length, for brevity
  integer :: day_of_year
  real :: temperature

  read*, station_name, day_of_year, temperature
  ! provide feedback (echo input)
  print*, "station_name=", trim(adjustl(station_name)), &
    " , day_of_year=", day_of_year, &
    " , temperature=", temperature
end program read_3variables_on_a_line

```

**Listing 2.7** src/Chapter2/read\_3variables\_on\_a\_line.f90

providing as input:

```
'Bremerhaven/Germany' 125 8<Enter>
```

On the other hand, if the input is performed as in the following program:

```

program read_3variables_on_3lines
  implicit none
  character(len=100) :: station_name ! fixed-length, for brevity
  integer :: day_of_year
  real :: temperature

  read*, station_name
  read*, day_of_year
  read*, temperature
  ! provide feedback (echo input)
  print*, "station_name=", trim(adjustl(station_name)), &
    " , day_of_year=", day_of_year, &
    " , temperature=", temperature
end program read_3variables_on_3lines

```

**Listing 2.8** src/Chapter2/read\_3variables\_on\_3lines.f90

we need to split the data for the three variables over three lines (records), as in:

```

'Bremerhaven/Germany' <Enter>
125<Enter>
8<Enter>

```

As previously mentioned, this form of I/O is not recommended for anything but quick testing, because it is limited from two points of view:

1. *system-dependent format*: the system will ensure that all data is visible, but the outcome is frequently not satisfying, due to generous whitespace-padding, which may often decrease readability; we discuss how to resolve this issue in Sect. 2.4.2.
2. *fixed I/O-channels*: input is only accepted from the keyboard, and output will be re-directed to the screen.<sup>30</sup> This becomes counter-productive as soon as the volume of I/O increases; we discuss how to route the I/O-channels to files in Sect. 2.4.3.

<sup>30</sup> For C/C++ programmers: this is the Fortran equivalent to the `stdin`, `stdout` and `stderr` streams.

**Exercise 2** (*Emission temperature of the Earth*) The simplest *energy balance model* (EBM) for computing the emission temperature ( $T_e$ ) of the Earth (as observed from space) consists of simply equating the absorbed solar energy and the outgoing blackbody radiation (assumed isotropic). This gives (Marshall and Plumb [8]) the following equation:

$$T_e = \sqrt[4]{S_0 \frac{(1 - \alpha_p)}{4\sigma}}$$

where  $\sigma = 5.67 \times 10^{-8} \text{ Wm}^{-2} \text{ K}^{-4}$  is the Stefan-Boltzmann constant and, for present-day, the average Earth albedo  $\alpha = 0.3$  and the annually-averaged flux of solar energy incident on the Earth is  $S_0 = 1367 \text{ Wm}^{-2}$ .

Write a program which evaluates this equation, computing  $T_e$ . How does the result change if  $S_0$  is 30 % lower? What about increasing  $\alpha$  by 30 %?

### 2.4.2 Customizing Format-Specifications

Fortran allows precise control on how data is converted between the internal representation and the textual representation used for formatted I/O. This is achieved by specifying a *format specification*. In fact, the language provides three ways of specifying the format:

1. *asterisk* (`(*)`): this is what we have used so far. The effective format is platform-dependent.
2. *a character string expression (of default kind)*: this consists of a list of *edit descriptors*, as will be explained in this section.
3. *a statement label*: this allows writing the format on a separate, labeled statement—a technique that may be useful for structuring I/O statements better. However, we do not emphasize this option, since the same effect can be obtained with character strings.

The basic form of the *output* statement is:

```
print <format> [, <I/O list>]
```

Similarly, the *input* statement looks like:

```
read <format> [, <I/O list>]
```

The *format*-part, on which we focus in this section, is usually a character expression of the form:

```
'( edit_descriptor_1, edit_descriptor_2, ... )'
OR
"( edit_descriptor_1, edit_descriptor_2, ... )"
```

where each edit descriptor in the comma-separated list corresponds to one or more<sup>31</sup> item(s) in the *I/O list* of the statement.

The task of the edit descriptor is to precisely specify how data is to be converted from the internal representation to the character representation external to the program (or the other way around). Fortran supports three types of edit descriptors, which can be combined freely: *data*, *character string*, and *control*.

**Data edit descriptors:** This is the most important category, since it refers to the actual data-conversion process. Such edit descriptors are composed of combinations of characters and positive integers, as discussed shortly. In general, the numbers represent the lengths of the different components in the text representation on the external device side. *For output of numeric types, a set of asterisks is printed if these lengths are too small for representing the value.*

Fortran provides different types of edit descriptors, for each of the intrinsic types.<sup>32</sup> We present them below, using monospace-font for characters that need to be typed literally, and *italic*-font for variables to be replaced by integer values. *Note that characters like `−` (negation), `.` (decimal point) and `e` or `E` (marker for exponent), when they appear, are also accounted for in the values of the various field-width variables.*

- integer: either `iw` or `iw.m` may be used, where *w* specifies the width of the field, and *m* specifies that, on output, at least *m* digits should be printed *even if they are leading zeroes* (on input, the two forms are equivalent). Example:

```
integer :: id = 0, year=2012, month=9, day=1
integer, dimension(40) :: mask = 10
print*, "Enter ID (integer < 1000):"
read'(i3)', id
! echo id (with leading zeroes if < 100)
print'(i3)', id
! using multiple edit descriptors
print'(i4, i2.2, i2.2)', year, month, day
```

When the magnitude of the integers to be written is not known in advance, it is convenient to use the `i0` edit descriptor, which automatically sets the field-width to the minimum value that can still contain the output value<sup>33</sup>:

<sup>31</sup> It is possible, and sometimes useful, to have less edit descriptors than elements in the *I/O list*. In such situations, the edit descriptors are reused, while switching to a new record (for examples, see Sect. 2.6.5).

<sup>32</sup> Special facilities also exist for arrays and derived types. We discuss the former in Sect. 2.6.5, after introducing the corresponding language elements. For the latter, see Metcalf et al. [10].

<sup>33</sup> This form is highly recommended, as it relieves the programmer from bugs associated with manually selecting the field width (corrupted, asterisks-filled fields can occur if the number of digits in the number exceeds the expected field width). However, this makes the formatting of values variable, and may not be appropriate for applications where precise control of alignment is important (like compatibility with other programs, or for improving the clarity of the output). Also, note that this approach *does not work* for input (where `i0` would cause any input value to be set to zero).

```
print '(i0)', testInt ! works correctly for any value
```

**Binary, octal, and hexadecimal (hex) integers:** For some applications, it can be useful to read/write integer-values in a non-decimal numeral system (bases 2, 8, and 16 being the most frequent). This is easily achieved in Fortran, by replacing the `i` with `b` (for binary), `o` (for octal) and `z` (for hexadecimal) respectively. The field-width can also be specified or auto-determined, just like when using the decimal base. The following program uses such edit descriptors to convert decimal values to other bases (some new elements in the program will be covered later):

```
program int_dec_2_other_bases
  implicit none
  integer :: inputInteger

  ! elements of this will be clarified later
  write(*,'(a)', advance='no') "Enter an integer: "
  ! get number (field width needs to be manually-specified)
  read '(i20)', inputInteger
  ! (string in format discussed later) print...
  print '( "binary: ", b0)', inputInteger !...min-width binary
  print '( "octal : ", o0)', inputInteger !...min-width octal
  print '( "hex   : ", z0)', inputInteger !...min-width hex
end program int_dec_2_other_bases
```

Listing 2.9 `src/Chapter2/int_dec_2_other_bases.f90`

- **real:** no less than seven types of edit descriptors are available for this type (reflecting Fortran’s focus on numerical computing): `fw.d`, `ew.d`, `ew.dee`, `esw.d`, `esw.dee`, `enw.d`, `enw.dee`, where *w* denotes the total width of the field, *d* the number of digits to appear after the decimal point, and *e* (when present) the number of digits to appear in the exponent field.

The first type of edit descriptor (based on `f`) is appropriate when the domain of the values includes the origin, and does not span too many orders of magnitude (say,  $0.01 \lesssim x \lesssim 1000$ ). Otherwise, the `e`-variants, which use exponential notation, are usually preferred. The different `e`-variants were introduced for supporting various conventions for representing floating-point values used in different fields. The distinction lies mainly in the way they scale the exponent, which correlates to the range of the significant (= the rest of the number, after excluding the exponent). This is summarized in Table 2.2 below.

**Table 2.2** Prefixes for exponential notation in edit descriptors for `real`

| Prefix                          | Resulting range for absolute value of significant |
|---------------------------------|---|
| <code>e</code>                  | [0.1, 1.0)  |
| <code>en</code> (“engineering”) | [1, 1000)   |
| <code>es</code> (“scientific”)  | [1, 10)   |

Similar to `integer-values`, `w` can be set to zero when performing output, causing a minimum field-width to be selected, which can still contain the significant.<sup>34</sup> However, this is not allowed for the `[e]`-variants.

The following program demonstrates the effects of different edit descriptors for writing real-values:

```

program edit_descriptors_for_reals
  implicit none
  ! get kind for high-precision real
  integer, parameter :: QUAD_REAL = selected_real_kind(33,4931)
  real(kind=QUAD_REAL) :: testReal

  write(*,'(a)', advance='no') "Enter a real number: "
  read('f100.50)', testReal
  ! print with various edit-descriptors
  print '(a, f0.2 )', "f0.2      : ", testReal
  print '(a, f10.2 )', "f10.2     : ", testReal
  print '(a, f14.4 )', "f14.4     : ", testReal
  print '(a, e14.4 )', "e14.4     : ", testReal
  print '(a, e14.6e3 )', "e14.6e3  : ", testReal
  print '(a, en14.4 )', "en14.4   : ", testReal
  print '(a, en14.6e3 )', "en14.6e3 : ", testReal
  print '(a, es14.4 )', "es14.4   : ", testReal
  print '(a, es14.6e3 )', "es14.6e3 : ", testReal
end program edit_descriptors_for_reals

```

**Listing 2.10** `src/Chapter2/edit_descriptors_for_reals.f90`

- `complex`: can be formatted using pairs of edit descriptors for real values.
- `logical`: supports the `[lw]` edit descriptor, where `w` denotes the width of the field (if `w = 1`, `[T]` or `[F]` are supported, while `w = 7` enables support for the expanded notation of boolean values, i.e., `[.true.]` and `[.false.]`). According to the language standard, the width is mandatory.
- `character strings`: can be used with the `[a]` or `[aw]` edit descriptors, where the first form automatically determines the necessary width to contain the string in the I/O list. The second form allows manual specification of the width but, unlike the similar mechanism for numbers, the value is not invalidated with asterisks if the string in the I/O list is larger than `w`. Instead, the non-fitting part of the string on the right-hand side is simply truncated. Alternatively, if `w` is larger than the length of the string in the I/O list, the string will be right-justified.

All data edit descriptors can be preceded by a positive integer, when more values for which the same format is appropriate appear in the I/O list. This is particularly useful when working with arrays, as we will illustrate in Sect. 2.6.5.

**Control edit descriptors:** these do not assist in data I/O, but allow instructing the I/O system to perform other operations related to the alignment of output. We only discuss how to insert spaces and start a new line here (see Metcalf et al. [10] for other details).

To insert spaces in output, use the `[nx]` edit descriptor, where `n` represents the number of spaces to be inserted. Similarly, to start a new record (line) without issuing another output statement, use the `[n/]` edit descriptor, where `n` represents the number

<sup>34</sup> However, unlike `integer`, the value of `d` remains important even in this case, since truncation is usually inevitable when converting floating-point binary numbers to the decimal base.

of records to be marked as complete.<sup>35</sup> The following program uses these ideas, to print three character-strings and a logical value, where the first two strings are separated by two spaces, and three empty lines separate the second from the third string:

```
program mixing_edit_descriptors1
  implicit none
  logical :: convergenceStat = .true.

  print '(a, 2x, a, 4/, a, 11)', &
    "Simulation", "finished.", &
    "Convergence status = ", &
    convergenceStat
end program mixing_edit_descriptors1
```

**Listing 2.11** src/Chapter2/mixing\_edit\_descriptors1.f90

### NOTE

The idea of counts (also known as “repeat counts”) in front of edit descriptors is actually more general, since these can also appear in front of data edit descriptors (e.g. `'(10i0)'`), or even in front of groups of edit descriptors, surrounded by parentheses (e.g. `'(5(f8.2, x))'`). These are useful mostly when working with arrays, therefore we discuss them in more detail in Sect. 2.6.5.

**Character string edit descriptors:** we already presented cases when character strings were already present in the format specification itself. These are permitted (but only for output), and can be combined with other types of descriptors, leading to output statements like in the next program:

```
program mixing_edit_descriptors2
  implicit none
  integer :: myInt = 213
  real :: myReal = 3.14

  print '( "An integer: ", i3, /, "A real: ", f0.2 )', &
    myInt, myReal
end program mixing_edit_descriptors2
```

**Listing 2.12** src/Chapter2/mixing\_edit\_descriptors2.f90

which should look more natural to C programmers.<sup>36</sup>

**Managing format specifications:** In the examples presented so far, we have written the format specification next to the I/O statements, as a string constant. This can be inconvenient in several situations, for example:

- when the same format specification needs to be reused for many I/O statements (here, the approach we have illustrated so far would lead to code duplication—always a red flag)

<sup>35</sup> Note that, if the current record is not empty, the number of empty records inserted by such an edit descriptor is  $n - 1$ .

<sup>36</sup> In C, the equivalent statement would be: `printf("An integer: %3d\nA real: %0.2f\n", anInt, aFloat);`

- when some facts about the format are not known until actual program execution (here, the string constant would impose switching between various hard-coded formats)

Fortunately, format specifications can also be non-constant strings, constructed dynamically at runtime. This can be used to address both issues above.<sup>37</sup> The following program illustrates how such a specification can be used for multiple output statements:

```
program string_variable_as_format_spec
  implicit none
  integer :: a = 1, b = 2, c = 3
  real :: d = 3.1, e = 2.2, f = 1.3
  ! format-specifier to be reused (could also use deferred-length)
  character(len=*), parameter :: outputFormat = '(i0, 3x, f0.10)'

  print outputFormat, a, d
  print outputFormat, b, e
  print outputFormat, c, f
end program string_variable_as_format_spec
```

**Listing 2.13** src/Chapter2/string\_variable\_as\_format\_spec.f90

### 2.4.3 Information Pathways: Customizing I/O Channels

The I/O statements discussed in the previous sections used the standard I/O channels: we always assumed that input is directed from the keyboard, and output is appearing on the screen. However, Fortran also allows the use of other channels (files or even character-strings), as will be discussed in this section.

Any I/O-channel (e.g. keyboard, screen, or a file on disk) is mapped to a *unit*. To distinguish between the various channels, each *unit* is identified by an integer unit-number, which is either

- selected by the platform (usually “5” represents standard-input from keyboard, and “6” standard-output to screen), or
- specified by the programmer (examples of this shown later).

**General I/O-statements:** The simplified forms of the I/O statements discussed previously (`print` and `read`) do not support customization of I/O channels. To gain more control, the general I/O statements (`write` and `read`<sup>38</sup>) need to be used, which we introduce below:

```
! general form of input statement
read ([unit]=u [,fmt=fm1] [,iostat=statCode] [,err=lb11] [,end=lb12]) &
  [inputList]
! general form of output statement
write([unit]=u [,fmt=fm1] [,iostat=statCode] [,err=lb11]) [outputList]
```

<sup>37</sup> There is also the option to use `format`-statements, as we mentioned previously. However, their usefulness is limited to the first issue, which is why we chose not to describe them—see Metcalf et al. [10] for details.

<sup>38</sup> The general input statement has the same name as the simplified form, but observe the other differences.



As usual, the square brackets denote optional items. The unit-number (*u*) and the format specification (*fmI*) are the only mandatory items (optionally, they can be preceded by `unit=` and `fmt=` respectively, to improve readability). Both of these items can be set to `*`, to recover the particular forms of I/O we already presented:

```

program general_can_recover_special_io
  implicit none
  integer :: anInteger

  ! special forms, default formatting...
  read *, anInteger ! input
  print *, "You entered: ", anInteger ! output

  ! ...equivalent general forms, default formatting
  read (*, *) anInteger ! input
  write(*, *) "You entered: ", anInteger ! output

  ! special forms, custom formatting...
  read '(i20)', anInteger ! input
  print '("You entered: ", i0)', anInteger ! output

  ! ...equivalent general forms, custom formatting
  read (*, '(i20)') anInteger ! input
  write (*, '("You entered: ", i0)') anInteger ! output
end program general_can_recover_special_io

```

**Listing 2.14** `src/Chapter2/general_can_recover_special_io.f90`

**Expecting the unexpected: exception handling** The remaining (optional) parameters in the general I/O-statements (which we named in the examples above *statCode*, *lbl1* and *lbl2* for read / *statCode* and *lbl1* for write) help the program recover from various exceptional conditions. Since the success of these I/O statements depends on properties of data channels usually beyond the control of our programs, many things can go wrong, without being a program bug. For example, when trying to read from a file, the file may not exist, or our program may not have permission to read from it. Similarly, the program may try to write to a file for which it has no write-permission, or there may not be sufficient space on the external device to contain the output data.

If the error-handling parameters are omitted, any problems encountered during the I/O operations will cause the program to crash, which is acceptable for test programs. However, for “industrial-strength” programs that will be run by many users, it is a good idea to put these error-handling facilities to good use, for example to assist the users. The meaning of the optional parameters is summarized below:

- `iostat=statCode`: here, *statCode* is an integer which will be set to a value representing the status of the I/O operation (following the Unix tradition, zero means “no error”, while a non-zero value signals that an error occurred)
- `err=lbl1`: *lbl1* is the label<sup>39</sup> of a statement (within the same (sub)program), to which the program will jump if an error occurred during the I/O statement

<sup>39</sup> In Fortran, every statement can be given a *label*, which is simply a positive integer (of at most 5 digits), written before the statement. These provide “bookmarks” within the code, allowing the program to “jump” to that statement when necessary—either transparently to the user (when the jump results from error handling), or explicitly (using the controversial `go to` statement). *Please note that explicit jumps with `go to` are strongly discouraged, as they can quickly make programs difficult to understand!*

- `end=lbl2`: `lbl2` is the label of a statement (within the same (sub)program), to which the program will jump if an “end-of-file” condition will be met (for the read-statement)

The following program illustrates how these extra arguments may be used:

```

program read_with_error_recovery
  implicit none
  integer :: statCode=0, x

  ! The safeguarded READ-statement
  read(unit=*, fmt=*, iostat=statCode, err=123, end=124) x
  print '(a, 1x, i0)', "Received number", x
  ! Normal program termination-point, when no exceptions occur
  stop

123 write(*, '(a, 1x, i0)') &
     "READ encountered an ERROR! iostat =", statCode
  ! can insert here code to recover from error, if possible...
  stop
124 write(*, '(a, 1x, i0)') &
     "READ encountered an end-of-file! iostat =", statCode
  ! can insert here code to recover from error, if possible...
  stop
end program read_with_error_recovery

```

**Listing 2.15** `src/Chapter2/read_with_error_recovery.f90`

**Exercise 3 (Testing error recovery)** Compile the program listed above, and try providing different types of input data, to test how the error-handling mechanism works.

**Hints:** try providing (a) a valid integer-value, (b) a string and (c) an end-of-file character (on Unix: type CTRL+d).

**The three phases of I/O:** Working with external data channels in Fortran implies the following sequence of phases:

1. **establishing the link:** before the I/O system can use a `unit`, a link needs to be established and a unique unit-number assigned. For standard I/O (keyboard/screen), the channels are pre-connected by the Fortran runtime system, without any intervention from the programmer.

However, for all other cases the link has to be established explicitly, with the `open`-statement. From the programmer’s point of view, the most important effect of this statement is to associate a `unit`-number to the actual data channel. This number is necessary for the next steps (e.g. when the actual I/O takes place). Currently, there are two methods for performing this association:

- a. Until Fortran 2003, the programmer was responsible for explicitly selecting a positive integer-value for the `unit`-number. For working with ASCII files,<sup>40</sup> the `open`-statement would then commonly look like:

<sup>40</sup> Creating “binary” files is also possible, but we avoid discussing this, in favor of another format which is more appropriate in ESS, i.e., `netCDF` (see Sect. 5.2.2).

```

open([unit=]unitNum [, file=fileName] &
    [, status=statusString] [, action=actionString] &
    [, iostat=statCode] [, err=labelErrorHandling] &
    )

```

where:

- `unitNum` is a positive integer variable/constant, assigned by the programmer. This will be used by the actual I/O statements.
- `fileName` is a character-string, representing the actual name of the file in the file system.<sup>41</sup> This can be omitted only when `statusString` `=="scratch"` (which is useful for creating temporary files, managed by the system, and usually deleted when the program terminates).
- `statusString` is one of the following strings: "old", "new", "replace", "scratch" or "unknown" (=default). This can be used to enforce some assumptions related to the status of the file prior to opening it.
- `actionString` is one of the strings: "read", "write" or "readwrite". This is useful for limiting the set of I/O statements that can be used with the unit, which can help prevent bugs.
- `statCode` and `labelErrorHandling` have the same roles as `statCode` and `lbl2` in the preceding discussion on error-handling.

The following listing presents some examples:

```

10 integer :: statCode
11 real :: windUx=1.0, windUy=2.0, pressure=3.0
12
13 ! assuming file"wind.dat"exists, open it for reading, selecting
14 ! the value of 20 as unit-id; no error-handling
15 open(unit=20, file="wind.dat", status="old", action="read")
16
17 ! open file"pressure.dat"for writing (creating it if it does not
18 ! exist, or deleting and re-creating it if it exists), selecting
19 ! the value of 21 as unit-id; place in variable 'statCode' the
20 ! result of the open-operation
21 open(unit=21, file="pressure.dat", status="replace", &
22     action="write", iostat=statCode)
23
24 ! open a scratch-file, for storing some intermediate-result (which
25 ! we need to read later), that would be too large to keep in memory;
26 ! no error-handling
27 open(unit=22, status="scratch", action="readwrite")

```

**Listing 2.16** `src/Chapter2/file_io_demo_manual_unit_numbers.`

`f90` (excerpt)

- Requiring the programmer to manually manage the unit-numbers (the “magic” numbers 20, 21, and 22 in the listing above) is inconvenient, especially for large projects. Fortunately, since Fortran 2008, it is possible to ask the runtime system to automatically provide a suitable unit-number, so that clashes with any other open links are avoided. The syntax for the open-statement is similar to the one previously shown, except that we need to replace `[unit=]unitNum` with `[newunit=]unitVariable`:

<sup>41</sup> Note that there might be some system-dependent restrictions on what constitutes a valid filename.

```

open([newunit=]unitVariable [, file=fileName] &
     [, status=statusString] [, action=actionString] &
     [, iostat=statCode] [, err=labelErrorHandling] &
     )

```

Note that, with this new method, it is not possible anymore to use constants for the `newunit`-value—only integer variables are accepted. This is because, when the `open`-statement is invoked, the runtime system will need to update `unitVariable`.<sup>42</sup>

With this new method, the examples presented above can be re-written as:

```

13  integer :: statCode, windFileID, pressureFileID, scratchFileID
14  real :: windUx=1.0, windUy=2.0, pressure=3.0
15  ! assuming file'wind.dat'exists, open it for reading, and store an
16  ! (automatically-acquired) unit-number in variable'windFileID'; no
17  ! error-handling
18  open(newunit=windFileID, file="wind.dat", status="old", &
19       action="read")
20
21  ! open file'pressure.dat'for writing (creating it if it does not
22  ! exist, or deleting and re-creating it if it exists), while storing
23  ! the (automatically-acquired) unit-number in variable'pressureFileID';
24  ! place in variable'statCode' the result of the open-operation
25  open(newunit=pressureFileID, file="pressure.dat", status="replace", &
26       action="write", iostat=statCode)
27
28  ! open a scratch-file, storing the (automatically-acquired) unit-number
29  ! in variable'scratchFileID'; no error-handling
30  open(newunit=scratchFileID, status="scratch", action="readwrite")

```

**Listing 2.17** `src/Chapter2/file_io_auto_manual_unit_numbers.`

£90 (excerpt)

### Good practice

Due to its convenience, we recommend to use this second method (using `newunit`) when opening files. We also rely on this technique in the later examples for this book (especially in Chap. 4).

- actual I/O calls:** the second phase corresponds to issuing the actual I/O-statements, for the data we want to read or write. We discussed this in the previous sections; the only change necessary for file I/O is that the `*` used until now for the `unit-id` needs to be replaced by the appropriate variable, as associated in advance within the `open`-statement. For example (continuing the example from the previous listing):

```

32  ! ... some code to compute pressure ...
33  read(windFileID, *) windUx, windUy
34
35  ! display on-screen the values read from the'wind.dat'-file
36  write(*, '(windUx =*, 1x, f0.8, 2x,windUy =*, 1x, f0.8)') &
37      windUx, windUy

```

<sup>42</sup> The standard specifies that a negative value (but different from `-1`, which signals an error) will be chosen for `unitVariable`, to avoid clashes with any existing code that uses the previous method of assigning `unit`-numbers, where positive numbers had to be used.

```

38      ! write to scratch-file (here, only for illustration-purpose; this makes
39      ! more sense if 'pressure' is a large array, which we would want to modify,
40      ! or deallocate afterwards, to save memory)
41      write(scratchFileID, '(f10.6)') pressure ! write to scratch
42      ! re-position file cursor at beginning of the scratch-file
43      rewind scratchFileID
44      ! ... after some time, re-load the 'pressure'-data from the scratch-file
45      read(scratchFileID, '(f10.6)') pressure
46
47      ! write final data to 'pressure.dat'-file
48      write(pressureFileID, '(f10.6)') pressure*2
49

```

**Listing 2.18** src/Chapter2/file\_io\_auto\_manual\_unit\_numbers.

f90 (excerpt)

3. **closing the link:** unlike the first phase (establishing the link), the system will automatically close the link to any active unit, if the program completes normally. It is, however, still recommended for the programmer to perform this step manually, to avoid losing data in case an exception occurs.<sup>43</sup> To terminate the link to a unit, the `close`-statement can be used:

```

close([unit=]unitNum [, status=statusString]
      [, iostat=statCode] [, err=labelErrorHandling]
)

```

Like for the `open`-statement, `unitNum` is mandatory, but some additional (optional) parameters are also supported:

- `statusString` can be either "keep" (=default, if the unit does not correspond to a scratch file) or "delete" (=required value for scratch files)
- `statCode` and `labelErrorHandling` can be used for error-handling, like for the `open`-statement

For example, the files opened in the previous listings can be closed with:

```

52      close(windFileID); close(pressureFileID); close(scratchFileID)

```

**Listing 2.19** src/Chapter2/file\_io\_auto\_manual\_unit\_numbers.

f90 (excerpt)

**Internal files:** In addition to units, the general I/O statements in Fortran can also operate on internal files (which are simply buffers, stored as strings or arrays of strings).<sup>44</sup>

Internal files are similar, in a sense, to the scratch files that we described earlier, since they are normally used for temporarily holding data which need to be manipulated at a later stage of the program's execution. However, because they are resident in

<sup>43</sup> Such data loss can occur when writing to files, since most platforms use buffering mechanisms for temporarily storing output data, to compensate for the slow speed of the permanent storage devices (e.g. disks).

<sup>44</sup> Strictly speaking, these do not form true I/O operations (the buffers are still memory areas associated with the program, so no external system is involved), but it is convenient to treat them as such (as done for the equivalent `stringstream` class in C++).

memory, they are usable only for smaller amounts of data. One application of internal files is type conversion between numbers and strings—for example, to dynamically construct names for the output files of an iterative model, at each time step.<sup>45</sup> One approach to achieve this is shown in the listing below:

```

1  program timestep_filename_construction
2  implicit none
3  character(40) :: auxString ! internal file (=string)
4  integer :: i, numTimesteps = 10, speedFileID
5
6  ! do is for looping over an integer interval (discussed soon)
7  do i=1, numTimesteps
8      ! write timestep into auxString
9      write(auxString,'(i0)') i
10     ! open file for writing, with custom filename
11     open(newunit=speedFileID, &
12          file="speed_"// trim(adjustl(auxString)) // ".dat", &
13          action="write")
14
15     ! here, we would have model-code, for computing the speed and writing
16     ! it to file...
17
18     close(speedFileID)
19 end do
20 end program timestep_filename_construction

```

**Listing 2.20** src/Chapter2/timestep\_filename\_construction.f90

**Non-advancing I/O:** We illustrated towards the end of Sect. 2.4.1 how, unlike other languages, Fortran automatically advances the file-position with each I/O statement, to the beginning of the next record. However, this can be turned off for a particular I/O-statement, by setting the optional control specification `advance` to "no" (default value is "yes"). This is often used when data is requested from the user, in which case it is desirable to have the prompt and the user input on the same line. We already used this technique, in Listings 2.9 and 2.10.

### 2.4.4 The Need for More Advanced I/O Facilities

So far, we discussed some basic forms of I/O, which are useful in common practice. However, these approaches do not scale well to the data throughput of state of the art ESS models (currently, in the terrabyte range for high-resolution models with global coverage). Text ("formatted") files are ineffective for handling such amounts of data, since each character in the file still occupies a full byte. If we imagine a very simple file which only contains the number 13, the ASCII-representation will occupy 2 bytes = 16 bits. In addition, to mark the end of each record, a newline character (Unix) or carriage-return + newline (Windows) needs to be added for every row in the file. Thus, the total space requirement for storing our number in a file will be of 3 bytes on Unix, and 4 bytes on Windows systems, respectively.

<sup>45</sup> Here, we imply there is one output file for each time step, to illustrate the idea. Note, however, that this may not always be a good approach. In particular, when the number of time steps is large, it is more convenient to write several time steps in each file (this is supported by the `netCDF`-format, which we will describe in Sect. 5.2.2).

Alternatively, if we choose to store the data directly in binary form, 4 *bits* would already be sufficient in theory to represent the number 13 (however, this is half of the smallest unit of storage—on most systems, the file would finally occupy 1 *byte*). These calculations illustrate that there is a large potential for reducing the final size of the files, even without advanced compression algorithms, just by storing data in the binary format instead of the ASCII representation. Other advantages include:

- *less CPU-time spent for I/O operations*: the conversion to/from ASCII also increases the execution time of the program, by an amount that can become comparable to the time spent for actual computations
- *approximation errors*: especially when working with floating-point data, approximation errors can be introduced each time a conversion between binary and ASCII representations takes place

While the benefits of binary storage are significant, it does have the problem that interpretation of data is made more difficult.<sup>46</sup> The importance of this cannot be overstated, which is why *it is not recommended* to use the binary format directly in most cases: a much more convenient solution in ESS is to use the `netCDF` data format, which allows efficient storage in a platform-independent way. We cover this topic later, in Sect. 5.2.2, after introducing some more language features.

## 2.5 Program Flow-Control Elements (`if`, `case`, Loops, etc.)

Most programs shown so far consisted of instructions that were executed in sequence. However, in real applications it is often necessary to break this ordering, as some blocks of instructions may need to be executed (once or several times) only when certain conditions are met. The generic name for such constructs is (*program*) *flow-control*, and Fortran has several of them, as we discuss in this section.

**Style recommendation:** In the examples below, we indent each block of program instructions, to clearly reflect situations when their execution is conditioned by a certain flow-control construct. Indentation is not required by the language (the compiler eventually removes whitespace anyway), but it *greatly* improves the clarity of the code, especially when multiple flow-control constructs are nested. *We highly recommend this practice.*

### 2.5.1 `if` Construct

The simplest form of flow-control can be achieved with the `if`-statement which, in its most basic form, executes a block of code only when a certain scalar logical condition is satisfied. This is illustrated by the following program, which asks for a number, and informs the user in case it is odd:

---

<sup>46</sup> Various technicalities (such as platform dependence of the internal, bit-level representation of the same data) can make the data transfer nontrivial for binary data.

```

program number_is_odd
  implicit none
  integer :: inputNum

  write(*, '(a)', advance="no") "Enter an integer number: "
  read(*, *) inputNum
  ! NOTE: mod is an intrinsic function, returning the remainder
  ! of dividing first argument by the second one (both integers)
  if( mod(inputNum, 2) /= 0 ) then
    write(*, '(i0, a)') inputNum, " is odd"
  end if
end program number_is_odd

```

Listing 2.21 src/Chapter2/number\_is\_odd.f90

In this case (when there is only one branch in the `if`), the corresponding code can be made even more compact, on a single line<sup>47</sup>:

```

if( mod(num, 2) /= 0 ) write(*, '(i0, a)') num, " is odd"

```

We may wish to extend the previous example, such that a message is printed also when the number is even. This can also be achieved with `if`, which supports an (optional) `else`-branch:

```

if( mod(num, 2) /= 0 ) then
  write(*, '(i0, a)') num, " is odd"
else
  write(*, '(i0, a)') num, " is even"
end if

```

Sometimes, if the primary logical condition of the `if`-construct is `.false.`, we may need to perform additional tests. This is still possible using `if` only, in the most general form of the construct, which introduces `else if` branches:

```

if( <logical_condition1> ) then
  ! block of statements for "then"
else if( <logical_condition2> ) then
  ! block of statements for first "else if" branch
else if( <logical_condition3> ) then
  ! block of statements for second "else if" branch
else
  ! block of statements if all logical conditions
  ! evaluate to .false.
end if

```

To illustrate, assume that we need to extend our previous example such that, when the number is even, we inform the user if it is zero. This can be implemented as in:

```

if( mod(num, 2) /= 0 ) then
  write(*, '(i0, a)') num, " is odd"
! num is odd, now check if it is zero
else if( num == 0 ) then
  write(*, '(i0, a)') num, " is zero"
! default, "catch-all" branch, if all tests fail
else
  write(*, '(i0, a)') num, " is non-zero and even"
end if

```

<sup>47</sup> Note that the keywords `then` and `end if` do not appear in the compact form.



Other constructs (including other if-statements) can appear within each of the branches of the conditional.<sup>48</sup> It is recommended to moderate this practice (since it can easily lead to code that is hard to follow), but sometimes it cannot be avoided. In such cases, proper indentation becomes crucial. Also helpful is the fact that Fortran allows ifs (as well as the rest of the flow-control constructs) to be named, to make it clear to which construct a certain branch belongs; when names are used, the branches need to bear the same name as the parent construct. This is illustrated in the following (artificial and a little extreme) example, which asks the user for a 3-letter string, and then reports the corresponding northern hemisphere season<sup>49</sup>:

```

program season_many_nested_ifs
implicit none
character(len=30) :: line
write(*,'(a)', advance="no") "Enter 3-letter season acronym: "
read(*,'(a)') line

if( len_trim(line) == 3 ) then
  winter: if( trim(line) == "djf" ) then
    write(*,'(a)') "Season is: winter"
  else if( trim(line) == "DJF" ) then winter
    write(*,'(a)') "Season is: winter"
  else winter
    spring: if( trim(line) == "mam" ) then
      write(*,'(a)') "Season is: spring"
    else if( trim(line) == "MAM" ) then spring
      write(*,'(a)') "Season is: spring"
    else spring
      summer: if( trim(line) == "jja" ) then
        write(*,'(a)') "Season is: summer"
      else if( trim(line) == "JJA" ) then summer
        write(*,'(a)') "Season is: summer"
      else summer
        autumn: if( trim(line) == "son" ) then
          write(*,'(a)') "Season is: autumn"
        else if( trim(line) == "SON" ) then autumn
          write(*,'(a)') "Season is: autumn"
        else autumn
          write(*,'(5a)') &
            "' ", trim(line), "' is not a valid acronym", &
            " for a season!"
        end if autumn
      end if summer
    end if spring
  end if winter
else
  write(*,'(5a)') &
    "' ", trim(line), "' is cannot be a valid acronym", &
    " for a season, because it does not have 3 characters!"
end if
end program season_many_nested_ifs

```

**Listing 2.22** `src/Chapter2/season_many_nested_ifs.f90`

Note that, while indentation and naming of constructs are helpful, the resulting code still looks complex, which is why we do not recommend including such extreme forms of nesting in real applications. For the current example, there is a way to simplify the logic using the case-construct, discussed next.

**Note on spacing:** In Fortran, several keywords (especially for marking the termination of a flow-control construct) can be written *with* or, equivalently, *without* spaces

<sup>48</sup> The process is called *nesting*. When used, nesting has to be complete, in the sense that the “parent”-construct must include the “child”-construct entirely (it is not allowed to have only partial overlap between the two).

<sup>49</sup> This is a common convention in ESS, where DJF = winter, MAM = spring, JJA = summer, and SON = autumn (for the northern hemisphere). The acronyms are obtained by joining the first letters of the months in each season.

in between. For example, `endif` is equivalent to `end if`, and `enddo` (discussed later)—to `end do`. This is more a matter of developer preferences.

## 2.5.2 *case Construct*

Another flow-control construct is `case`, which allows comparing an expression (of logical, integer, or character type) against different values and ranges of values. The general syntax for it is:

```
select case( <expression> )
case ( <match_list1> )
  ! block of statements when expression evaluates to
  ! a value in match_list1

case ( <match_list2> )
  ! block of statements when expression evaluates to
  ! a value in match_list2

! ... (other cases)

case default
  ! block of statements when no other match was found
  ! ("catch-all" case)
end select
```

Unlike the `if`-construct, where multiple expressions could be evaluated by adding `else if`-branches, `case` only evaluates one expression, and afterwards tries to match this against each of the cases. To avoid ambiguities, the patterns in the different match-lists are *not* allowed to overlap.

Note that only (literal) constants are allowed in each match-list. An interesting feature related to the match-list is that ranges of values are allowed (for types `integer` and `character`). Furthermore, values and ranges can be combined freely. This is shown in the following example, which reads a character, and tests if it is a vowel (assuming the English alphabet):

```
program vowel_or_consonant_select_case
implicit none
character :: letter

write(*,'(a)', advance="no") &
  "Type a letter of the English alphabet: "
read(*,'(a1)') letter

select case( letter )
case ('a','e','i','o','u', &
      'A','E','I','O','U')
  write(*,'(4a)') " ", letter, " ", " is a vowel"
  ! note below: match-list consists of values,
  ! as well as value-ranges
case ('b':'d','f','g','h','j':'n','p':'t','v':'z', &
      'B':'D','F','G','H','J':'N','P':'T','V':'Z')
  write(*,'(4a)') " ", letter, " ", " is a consonant"
case default
  write(*,'(4a)') " ", letter, " ", " is not a letter!"
end select
end program vowel_or_consonant_select_case
```

**Listing 2.23** `src/Chapter2/vowel_or_consonant_select_case.f90`

For specifying ranges of values, it is even allowed to omit the lower or the higher bound (but not both), which allows ranges to extend to the smallest (negative) and largest (positive) representable integer-value.<sup>50</sup> This is used in the next code listing, which asks the user to enter an integer value, and checks if the number is a valid index for a calendar month:

```

program check_month_index_select_case_partial_ranges
implicit none
integer :: month
write(*, '(a)', advance="no") "Enter an integer-value: "
read(*, *) month

! check if month is valid month-index, with partial
! (semi-open) ranges in a select-case construct
select case( month )
case ( :0, 13:)
write(*, '(a, i0, a)') "error: ", &
month, " is not a valid month-index"
case default
write(*, '(a, i0, a)') "ok: ", month, &
" is a valid month-index"
end select
end program check_month_index_select_case_partial_ranges

```

**Listing 2.24** src/Chapter2/check\_month\_index\_select\_case\_partial\_ranges.f90

Using the case-construct can lead to great simplifications of what would otherwise be complex, nested if-contraptions. For example, the season-acronym matching program, could be re-written as:

```

program season_select_case
implicit none
character(len=30) :: line

write(*, '(a)', advance="no") "Enter 3-letter season acronym: "
read(*, '(a)') line

if( len_trim(line) == 3 ) then
season_match: select case( trim(line) )
case ("djf", "DJF") season_match
write(*, '(a)') "Season is: winter"
case ("mam", "MAM") season_match
write(*, '(a)') "Season is: spring"
case ("jja", "JJA") season_match
write(*, '(a)') "Season is: summer"
case ("son", "SON") season_match
write(*, '(a)') "Season is: autumn"
case default season_match
write(*, '(5a)') &
' ', trim(line), ' ', " is not a valid acronym", &
" for a season!"
end select season_match
else
write(*, '(5a)') &
' ', trim(line), ' ', " is cannot be a valid acronym", &
" for a season, because it does not have 3 characters!"
end if

end program season_select_case

```

**Listing 2.25** src/Chapter2/season\_select\_case.f90

where we also demonstrated how to assign a name (in this example: season\_match) to the case-construct.

<sup>50</sup> These are, in a sense, the discrete equivalents of  $\pm\infty$ .

### 2.5.3 *do Construct*

The flow-control constructs discussed so far (*if* and *case*) allow us to determine whether blocks of code need to be executed or not. Another pattern, which is extremely important in modeling, is to execute certain blocks of code *repeatedly*, until some termination criterion is satisfied. This pattern (also known as *iteration*) is supported in Fortran through the *do*-construct, which we describe in this section.

The simplest form of iteration uses an integer-counter, as in the following example:

```
integer :: i
do i=-15, 10
  ! block of statements, to be executed for each iteration
  write(*, '(i0)') i
end do
```

Here, the variable *i* is also known as the *loop counter*, and needs to be of *integer* type. The numbers on *line 2* represent the lower (−15) and upper bound (10). For each value in this range, the block of statements within the *do*-loop will be executed. Within this block, the value of *i* can be *read* (e.g. it can appear in expressions), but it *cannot* be *modified*.

#### 2.5.3.1 Loop Counter Increment

By default, the loop counter is incremented by one at the end of each iteration. Fortran also allows to specify a different increment, as a third number at the beginning of the *do*-construct. This allows, for example, incrementing the loop counter in larger steps, or even decrementing it, to scan the range of numbers backwards. For example:

```
! iterate from 0 to 100, in steps of 25
do i=0, 100, 25
  ! block of statements
end do

! iterate backward, from 8 to -8, in steps of 2
do i=8, -8, -2
  ! block of statements
end do
```

In our examples so far, we always used integral literals for the *start*-, *end*-, and *increment*-values of the loop counter. However, the language also allows these to be integer-variables, or even more complex expressions involving variables. In such cases, the variables can be altered within the loop, but this has no influence whatsoever on the progress of the loop, since only the initial values are used for “planning” the loop. For example, in the following listing, the assignments on *lines 6* and *7* have no impact on the loop:

```

1 program do_specified_with_expressions
2   implicit none
3   integer :: timeMax = 10, step = 1, i, numLoopTrips = 0
4
5   do i=1, timeMax, step
6     timeMax = timeMax / 2
7     step = step * 2
8     numLoopTrips = numLoopTrips + 1
9     write(*, '(a, i0, a, /, 3(a, i0, /))') &
10      "Loop body executed ", numLoopTrips, " times", &
11      " i = ", i, &
12      " timeMax = ", timeMax, &
13      " step = ", step
14   end do
15
16 end program do_specified_with_expressions

```

**Listing 2.26** src/Chapter2/do\_specified\_with\_expressions.f90

**Exercise 4** (*Practice with do-loops*) The equidistant cylindrical projection is one of the simplest methods to visualize the Earth surface in a plane. This projection maps meridians and parallels onto vertical and horizontal lines, respectively. However, this projection is not “equal area”—for example, axis-aligned rectangles (say,  $10^\circ$  latitude  $\times$   $10^\circ$  longitude) which have the same area on the map do not have equal areas in reality.

To quantify this effect, use a `do`-loop to evaluate areas of 9 such cells (with latitude bounds  $[0^\circ N, 10^\circ N]$ ,  $[10^\circ N, 20^\circ N]$ , ...  $[80^\circ N, 90^\circ N]$ . How large is the area of the near-pole cell, relative to that of the near-equator cell (in percents)?

**Hint:** Assuming our vertical displacement is much smaller than the average Earth radius, a “cell” whose normal coincides with the local direction of gravity has an area given approximately by:

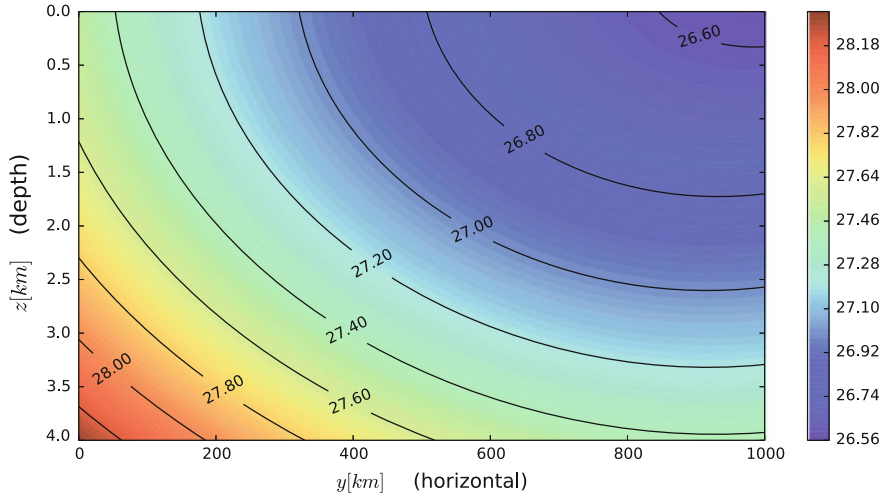
$$S_i = R_E^2 (\lambda^E - \lambda^W) (\sin \phi^N - \sin \phi^S),$$

where both latitudes ( $\lambda^{[E,W]}$ ) and longitudes ( $\phi^{[S,N]}$ ) are given in radians.

**Exercise 5** (*Hypothetical potential density profile*) Assume the potential density profile for a rectangular box within the ocean is given by:

$$\sigma_\theta(y, z) = \left[ 0.9184 \left( -\sqrt{G(y, z)} + 1 \right)^2 + 0.9184 \arccos^2 \left( \frac{1}{\sqrt{G(y, z)}} \left( 2 - \frac{y}{H} \right) \right) + 26.57 \right] \text{ kg/m}^3 \quad (2.1)$$

$$G(y, z) = \left( 2 - \frac{y}{H} \right)^2 + \left( 0.1 + \frac{z}{H} \right)^2 \quad (2.2)$$



**Fig. 2.1** Idealized profile of potential density ( $\sigma_\theta$ ), based on Eqs. (2.1)–(2.2)

where:

- $y \in [0, L]$ , with:  $L = 1000$  km
- $z \in [0, H]$ , with:  $H = 4$  km

This can be viewed as an idealized profile of the density structure in some part of the ocean (Fig. 2.1).

Assuming the extent along the  $x$ -axis (perpendicular to the figure) is of 100 km, compute the fraction of total volume occupied by water whose potential density matches the range typical for upper Labrador Sea Water (uLSW), which is:

$$\sigma_\theta^{\text{uLSW}} \in [27.68, 27.74] \text{ kg m}^{-3}$$

### 2.5.3.2 Non-deterministic Loops

In practical applications, loops are not always deterministic.<sup>51</sup> Suppose we need to read successive data elements (e.g. a time series) from a file, for estimating the mean and the variance of the values. The steps of the algorithm are the same for each considered value, so it is natural to surround them by a loop construct. However, since the data resides in the external file, we may not know in advance how many values there are. Fortran accommodates such cases with the “endless” `do` construct, which looks like:

<sup>51</sup> “Non-deterministic” means, in this context, *not (easily) determined at compilation time*.

```
do
  ! block of statements
end do
```

This form truly has the tendency to run endlessly,<sup>52</sup> and it is the responsibility of the programmer to devise a suitable termination criterion, and to end the execution of the loop with the `exit`-statement. This is illustrated in the following listing, which demonstrates a way to solve the file-reading problem described above, where a suitable loop-termination criterion is that the end-of-file was reached while trying to read-in data:

```
1 program mean_and_standard_deviation_from_file
2 implicit none
3 integer :: statCode, numVals=0, inFileID
4 real :: mean=0.0, variance=0.0, sd=0.0, newValue, &
5       sumVals=0.0, sumValsSqr=0.0
6
7 ! open file for reading
8 open(newunit=inFileID, file="time_series.dat", action="read")
9
10 ! "infinite" DO-loop, to read an unknown amount of data-values
11 data_reading_loop: do
12   read(inFileID, *, iostat=statCode) newValue
13   ! check if exception was raised during read-operation
14   if( statCode /= 0 ) then ! **TERMINATION-CRITERION for DO-loop**
15     exit data_reading_loop
16   else ! datum read successful
17     numVals = numVals + 1
18     sumVals = sumVals + newValue
19     sumValsSqr = sumValsSqr + newValue**2
20   end if
21 end do data_reading_loop
22
23 ! close file
24 close(inFileID)
25
26 ! evaluate mean (avoiding division by zero, when file is empty)
27 if( numVals > 0 ) mean = sumVals / numVals
28 ! evaluate 2nd central-moment (variance)
29 variance = (sumValsSqr - numVals*mean**2) / (numVals - 1)
30 ! evaluate standard-deviation from variance
31 sd = sqrt( variance )
32
33 write(*, '(2(a, f10.6))') "mean = ", mean, &
34       ", sd = ", sd
35 end program mean_and_standard_deviation_from_file
```

**Listing 2.27** src/Chapter2/mean\_and\_standard\_deviation\_from\_file.f90

where we used the fact that:

$$s\{X\} \equiv \sqrt{\text{var}\{X\}} = \sqrt{\frac{\sum_{i=1}^N (x_i - \bar{x})^2}{N-1}} = \dots = \sqrt{\frac{1}{N-1} \left( \sum_{i=1}^N x_i^2 - N\bar{x}^2 \right)}$$

where  $s$  is the unbiased estimator of the standard deviation,  $N$  is the number of samples,  $\bar{x}$  is the estimated mean, and  $x_{i \in [1..N]}$  corresponds to the individual samples.

If the loop that we wish to terminate is named, it is possible to provide this name to the `exit`-statement, to improve the clarity of the code. We illustrated this in the example above, although the value of this feature is more obvious when several loops are nested.

<sup>52</sup> At least, until the program is terminated forcibly.

### 2.5.3.3 Shortcutting Loops

Another pattern that occurs sometimes while working with loops is skipping over parts of the code within the loop's body, when certain conditions are met, *without leaving the loop*. For example, assume we are writing a program which converts a given number of seconds into a hierarchical representation (weeks, days, hours, minutes, and seconds). Clearly, the number of seconds provided by the user should be positive for the algorithm to work. If the user provides a negative integer, it does not make sense to try to find a hierarchical representation of the period; instead, it would be more useful to skip the rest of the code within the loop, and proceed to the next loop iteration directly, where the user has the opportunity to provide another input value. This type of behavior is supported in Fortran, using the `cycle [loop_name]`<sup>53</sup> command, as illustrated in the following example:

```

program do_loop_using_cycle
  implicit none
  integer, parameter :: SEC_IN_MIN = 60, &
    SEC_IN_HOUR = 60*SEC_IN_MIN, & ! 60 minutes in hour
    SEC_IN_DAY = 24*SEC_IN_HOUR, & ! 24 hours in a day
    SEC_IN_WEEK = 7*SEC_IN_DAY ! 7 days in a week
  integer :: secIn, weeks, days, hours, minutes, sec

  do
    write(*, '( /, a)', advance="no") & ! '/' adds newline, for separation
    "Enter number of seconds (or 0 to exit the program): "
    read(*, *) secIn

    if( secIn == 0 ) then ! loop-termination criterion
      exit
    else if( secIn < 0 ) then ! skipping criterion
      write(*, '(a)') "Error: number of seconds should be" // &
        " positive. Try again!"
      cycle ! ** calculation skipped with CYCLE **
    end if

    ! calculation using the value
    sec = secIn ! backup value
    weeks = sec / SEC_IN_WEEK; sec = mod(sec, SEC_IN_WEEK)
    days = sec / SEC_IN_DAY; sec = mod(sec, SEC_IN_DAY)
    hours = sec / SEC_IN_HOUR; sec = mod(sec, SEC_IN_HOUR)
    minutes = sec / SEC_IN_MIN; sec = mod(sec, SEC_IN_MIN)
    ! display final hierarchy
    write(*, '(i0, a)') secIn, "s = { ", &
      weeks, " weeks, ", days, " days, &
      hours, " hours, ", minutes, " minutes, ", &
      sec, " seconds }"

  end do
end program do_loop_using_cycle

```

Listing 2.28 `src/Chapter2/do_loop_using_cycle.f90`

Nesting of loops is another very common practice in ESS modeling, naturally occurring from the discretization of space and time. Another example of loop nesting occurs in linear algebra, for example matrix multiplication or transposition.

<sup>53</sup> `loop_name` is an optional name, which allows to clarify to which loop the `cycle-` command should be applied, in case of multiple nested `do`-loops.



**Exercise 6** (*Zero-padded numbers in filenames*) The program in Listing 2.20 produced filenames in which the numeric portion had a variable width. This may prevent some post-processing tools from correctly identifying the order of the files.

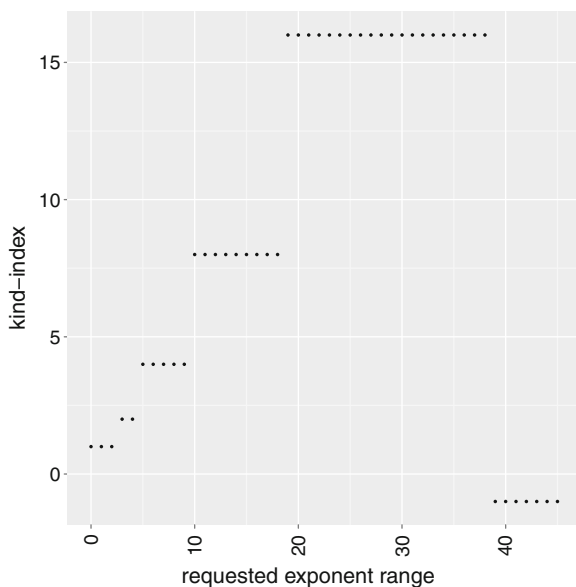
Extend the program, so that the numeric portion in filenames has a constant width (with zero-padding), which is calculated based on the value of `num_timesteps`.

**Hints:** if `num_timesteps` is zero, the required number of digits is obviously one; for the other cases, you can use the expression `aint(log10(real(num_timesteps)))+1` (we assume `num_timesteps`  $\geq 0$ ). Also, you can use a second internal file, to construct the format for the statement where the `i` is written to `aux_strng` (since a dynamic minimum width of the integer field needs to be specified).

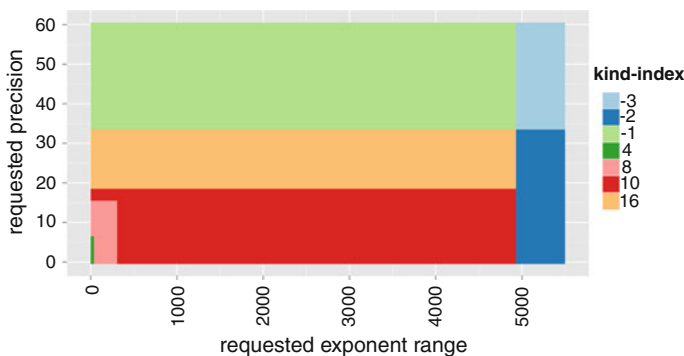
**Exercise 7** (*Detecting kinds of numeric types on your platform*) We now have the tools to complete the discussion on kind-values (Sect. 2.3.4). Write a program that uses the intrinsic functions `selected_int_kind` and `selected_real_kind` to determine the variants of these two numeric types available on your platform.

**Hints:** For each type, search the parameter space with do-loops. For `integer`, iterate through values for `requestedExponentRange` in the interval  $[0, 45]$ , and write to a file the (`requestedExponentRange`, `obtained_kind`)-pairs, as determined by your program. For `real`, use two nested do-loops, to iterate through values of `requestedExponentRange` in the interval  $[0, 5500]$ , and values of `requestedPrecision` in the interval  $[0, 60]$ , and write to another file the (`requestedExponentRange`, `requestedPrecision`, `obtained_kind`)-triplets. Visualize your results as a scatter plot for `integer` and a filled contour map for `real` (the results for our platform are shown in Figs. 2.2 and 2.3).

**Exercise 8** (*Working with another platform*) Use the program developed for the previous exercise to test the kind-values for a different platform (hardware and/or compiler). Compare the results with those obtained in Exercise 7.



**Fig. 2.2** integer kind indices as a function of requested exponent range (platform: Linux, 64 bit, gfortran compiler)



**Fig. 2.3** real kind indices as a function of requested exponent range and requested precision (platform: Linux, 64 bit, gfortran compiler)

## 2.6 Arrays and Array Notation

So far, we used mostly scalar variables for representing entities in our example programs. This was sufficient, since the number of quantities was rather limited. However, in most applications (and ESS models in particular), the number of variables easily exceeds several millions, which is clearly not something that can be managed with scalars. There is, in fact, a distinct branch in computer science, dealing with *data*

*structures*—methods of organizing data for various applications.<sup>54</sup> In this section, we focus on arrays, which are among the most basic, but also most popular data structures. In fact, arrays are so useful in scientific and engineering programs that a large part of the Fortran language is devoted to them.

An array is a compound object, which holds a set of elements. The elements can belong to any of the 5 intrinsic types already discussed, or even to derived types. An important constraint, however, is that all elements need to have the same type (and `kind-parameter`).

The second important aspect of arrays, besides the type of each element they store, is their shape. It is helpful to introduce some terms, which characterize this aspect for any given array:

- *rank* = number of dimensions of the array. “Dimensionality” in this context refers to the number of indices needed for uniquely specifying an element—similar to classification of tensors in mathematical physics.<sup>55</sup>
- *extent* = “width” along a particular dimension. Fortran arrays are *rectangular*, in the sense that the extent along each dimension is independent of the value of the indices along the other dimensions.<sup>56</sup> We will demonstrate later how the range for each index can be freely customized in Fortran, by specifying arbitrary (possibly negative) integers for the lower and upper bound. In this context, we have  $\text{extent} == \text{upper\_bound} - \text{lower\_bound} + 1$ .
- *shape* = 1D-array, each component of which represents an extent along a specific dimension.
- *size* = total number of scalar elements in the array (equals product of extents).

### 2.6.1 Declaring Arrays

Before working with arrays, we need to create them. This needs to be done explicitly in Fortran, and it implies *declaring* and *initializing* the arrays we want to use (second step is mandatory for constants, but highly recommended for modifiable arrays too).

In normal usage, there are two ways for declaring arrays, both of which require specification of the array *shape*. The first method uses the `dimension`-keyword, as in:

---

<sup>54</sup> Because the merits of a *data structure* can only be proven in the context of the *algorithms* applied on them, most references unify these two aspects (e.g. Mehlhorn and Sanders [9] or Cormen et al. [2]).

<sup>55</sup> At the risk of stating the obvious: this should not be confused with dimensionality of the physical space (if we store the components of a 3D-vector in an array, that array will have `rank==1`).

<sup>56</sup> So an entity with a more irregular shape, such as the set of non-zero elements of a lower-triangular matrix, needs to be stored indirectly when arrays are used.

```
! both X & Y are rank=1 arrays, with 16 elements each
real, dimension(16) :: X, Y
! A is a rank=3 array, with 520^3 elements
! up-to rank=15 is allowed in Fortran 2008 (was 7 in Fortran 90)
integer, dimension(520, 520, 520) :: A
```

The second declaration method is to specify the shape of the array after the variable name, as in:

```
! X is still a rank=1 array, but Y is a scalar real
real :: X(16), Y
! same effect as in previous declaration of A
integer :: A(520, 520, 520)
```

The numbers inside the shape specification actually represent the *upper bounds* for the indices along each dimension. An interesting feature in Fortran is that one can also specify *lower bounds*, to bring the code closer to the problem-domain:

```
real, dimension(-100:100) :: Z ! rank=1 array, with 201 elements
```

## Notes

- Unlike programming languages from the C-family, the value to which the lower bound defaults (when it is not specified) is **1 (not 0)**!
- Although in the examples here we often specify the shape of the arrays using hard-coded integer values, it is highly recommended to use named integer constants<sup>57</sup> for this in real applications, which saves a lot of work when the size of the arrays needs to be changed (since only the value of the constant would need to be edited).

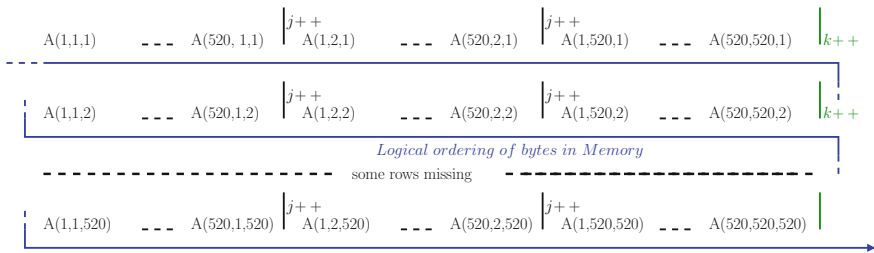
## 2.6.2 Layout of Elements in Memory

We now turn our attention to a seemingly low-level detail which is, however, crucial for parts of our subsequent discussion: *given one of the array declarations above, how are the array elements actually arranged in the system's memory*<sup>58</sup>?

The memory can be viewed as a very large 1D sequence of bytes, where all the variables associated to our program are stored. For 1D-arrays, it is only natural to store the elements of the array contiguously in memory. Things are more complex for arrays of `rank > 1`, where an ordering convention (also known as “array element order”) for the array elements needs to be adopted (effectively, defining a *mapping* from the tuple of coordinates in the array to a linear index in memory).

<sup>57</sup> This is achieved with the `parameter`-attribute.

<sup>58</sup> Here, we refer to the Virtual Memory subsystem, which includes mainly the *random-access memory* (RAM) and, less used nowadays, portions on secondary storage (e.g. hard-drives) which extend the apparent amount of memory available.



**Fig. 2.4** Illustration of element ordering for a 3D array in Fortran. The *dashed horizontal black line* represents incrementing in the first dimension, the *black vertical lines*—incrementing in the second dimension, and the *vertical green lines*—incrementing in the third dimension. The *blue line* represents the logical ordering of bytes in memory. The figure was split into multiple rows, to fit in the page

**NOTE**

In Fortran, the array element order for elements of a multi-dimensional array is such that the earlier dimensions vary faster than the latter ones.<sup>a</sup> *This is exactly opposite to the corresponding convention in C and C++, providing opportunities for bugs to appear while porting applications!*

<sup>a</sup> An alternative way to remember this is relative to how a matrix is stored: since the elements within a column are adjacent, Fortran (along with other languages like MATLAB and GNU Octave (octave)) is said to use *column-major order* (C and C++ use *row-major order*).

For example, the elements of the A-array declared earlier could be arranged in memory similarly to Fig. 2.4.

The array element order is important for understanding how several facilities of the language work with multi-dimensional arrays. It is also very relevant for application performance,<sup>59</sup> as illustrated in Exercise 9.

**2.6.3 Selecting Array Elements**

Since arrays group multiple elements, a crucial feature when working with them is the ability to select elements based on some pattern, which is usually dictated by a subtask of the algorithm to be implemented. Fortran supports many methods for

<sup>59</sup> This relates to the memory-hierarchy within modern systems. There are usually several layers of *cache*-memory (very fast, but with small capacity) between the CPU and RAM, to hide the relatively high latency for fetching data from RAM. Most caches implement a pre-fetching policy, and higher performance is achieved when the order in which array elements are processed is close to the array element order. Note that more details need to be considered, for performance-critical (sub)programs (for more information, see Hager and Wellein [5]).

outlining such selections. We illustrate these via examples below, assuming we want to overwrite some parts of an array. However, the same techniques apply for reading parts of an array, of course.

Given an array declaration like:

```
integer, parameter :: SZ_X=40, SZ_Y=80
! Note the use of named integer constants for specifying
! the shape of the array (recommended practice).
real, dimension(-SZ_X:SZ_X, -SZ_Y:SZ_Y) :: temperature
```

Fortran allows to select:

- **the entire array:** by simply specifying the array's name:

```
temperature = 0. ! scalar written to selection (=whole array)
```

- **a single element:** by specifying the array's name, followed, within parentheses, by a list of  $n$  indices<sup>60</sup> (where  $n$  is the rank of the array):

```
! scalar written to element (i=1, j=2)
temperature(1, 2) = 10.
```

- **a sub-array:** by specifying the array's name followed, within parentheses, by a list of  $n$  ranges ( $n$  = rank of the array, as before). A range, in this context, is an integer interval, with an optional step,<sup>61</sup> as in:

```
! scalar written to element (i=1, j=2)
temperature(-SZ_X:0, -SZ_Y:SZ_Y:2) = 20.
```

- **a list of elements:** by specifying the array's name followed, within parentheses, by one or more array(s) of `rank==1` (we call these *selection arrays*). Each selection array represents a list of values for a corresponding dimension (so only one selection array is necessary when the source array is 1D, two when the source array is 2D, etc.). The elements of the source array which eventually become selected are those with the coordinate-tuples within the Cartesian product of the sets represented by the selection arrays. The next listing uses this procedure to select the corners of the 2D-array `temperature`:

```
! only 4 elements are selected (Cartesian product):
! (-SZ_X, -SZ_Y), (-SZ_X, SZ_Y), (SZ_X, SZ_Y), (SZ_X, -SZ_Y)
temperature( [ -SZ_X, SZ_X ], [ -SZ_Y, SZ_Y ] ) = 30.
```

where we used the `[ ]` and `[ ]` tokens, to create arrays inline.<sup>62</sup> We will present more uses of this technique in the next section.

<sup>60</sup> The list of indices can also be provided as a 1D-array of size  $n$ .

<sup>61</sup> Such ranges are very similar to what we illustrated previously for the `do` program-flow construct, except that in this case commas (`,` ) need to be replaced by colons (`,` ).

<sup>62</sup> This notation was introduced in Fortran 2003. Note that there is also an older (equivalent) notation, using the tokens `( / )` and `( / )`.

**NOTE**

When an array selection is used for writing to an array, it is not recommended to have, in the selection arrays, elements which are repeated, since this can lead to attempts to write more than one value to the same array element.<sup>a</sup>

<sup>a</sup> Some compilers may allow this without warnings, although the standard declares these as illegal. In any case, the behavior in such situations is likely platform-dependent, and the recommendation holds.

### 2.6.4 Writing Data into Arrays

As soon as an array is declared, a first concern, before using the values of the array elements in other statements, is to initialize those values. Unlike other languages, the Fortran standard does not make any guarantee regarding data initialization (such as setting them to zero), so explicit action is required from the programmer in this respect.

Values can be assigned to array elements using several mechanisms, to fit various scenarios. Just as for scalar variables, these assignments can be combined<sup>63</sup> with the declaration line, as a compact method of initialization (therefore, the techniques shown in this section apply to initialization, as well as to assignment).

An important notion when writing data to an array is *conformability*: two data entities are said to be conformable if they are arrays with the same shape, or if at least one of them is a scalar. When one entity is assigned to another one, they need to be conformable (this is also necessary when forming array expressions, as discussed later).

#### 2.6.4.1 Writing a Constant Value

One of the simplest write operations is to assign a scalar value to an entire array (or an array section), in which case all elements (selected elements) will be set to that value:

```
! either: declaration, followed by assignment
! before the values are used
real, dimension(0:20) :: velocity
velocity = 0.
! or: initialization directly at declaration-time
real, dimension(0:20) :: velocity = 0.
```

<sup>63</sup> For array constants this is, naturally, required.

### 2.6.4.2 Writing Element-by-Element

Another form of writing into an array is the “lower-level” fashion, using element-based assignments, (optionally) combined with loops. This is the most flexible method and, perhaps, also the most intuitive. As a simple example, here is a more verbose (but logically equivalent) version of the assignment for the `velocity` array from the previous listing:

```
integer :: i
! element-based assignment (equivalent to: velocity = 0.)
do i=0,20
  velocity(i) = 0.
end do
```

Despite being conceptually straightforward, *we recommend avoiding this procedure* when possible, in favor of the ones discussed previously (writing a constant value), or next (writing another array(selection)). Still, this form is sometimes justified, for example when:

- the assignment does not follow an obvious pattern, or
- there is a definite performance advantage (proven by benchmarks) for using this method instead of the other ones.

### 2.6.4.3 Writing Another Array (Section)

An array (or array-section) can also be assigned to another array (or section), as long as the two entities are conformable. For example:

```
integer :: array1(-10:10), array2(0:20)
! ... some code to compute array2 ...
array1 = array2      ! whole-array assignment
```

Note that the arrays are conformable even if the lower and upper bounds of the array indices are different for the two arrays, as it was the case here (only the *shape* matters): after the assignment, `array1(-10) == array2(0) == ... == array1(10) == array2(20)`.

The use of array sections is illustrated in the following listing, which swaps the value of each odd element with that of the next even element<sup>64</sup>:

```
integer :: array3(1:20), tmpArray(1:10)
! ... some code to initialize array3 ...
tmpArray = array3(1:20:2)
array3(1:20:2) = array3(2:20:2)
array3(2:20:2) = tmpArray
```

<sup>64</sup> This assumes the lower bound for the index is odd, and that the upper one is even.



### 2.6.4.4 Array Constructors

We already mentioned that arrays can be initialized based on other arrays, but then one could ask how are the latter arrays to be initialized. Fortran has a special facility for this problem—the *array constructor*. This consists of a list of values, surrounded by square brackets.<sup>65</sup> A common use of this is to define a *constant* array (with the *parameter-keyword*), as in:

```
integer, dimension(3), parameter :: meshSize = [ 213, 170, 10 ]
real, dimension(0:8), parameter :: weights = [ 4./9., &
  1./9., 1./36., 1./9., 1./36., &
  1./9., 1./36., 1./9., 1./36. ]
```

The arrays defined with the constructor syntax can also be used directly in expressions (as long as they are conformable with the other components of the expression), as any other array, for example:

```
integer, dimension(10) :: xRange
xRange = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
```

### 2.6.4.5 Patterns in Array Constructors: Implied-do Loops

A drawback of the *weights* and *xRange* examples above (using constructor syntax) is that they tend to be quite verbose. The *implied-do loops* were introduced in Fortran to solve this problem, when the values follow a well-defined pattern. They act as a convenient shorthand notation, with the general form:

```
! Note: the expression below needs to be embedded into an
! actual array constructor (see next examples).
( expr1, expr2, ..., indexVar = exprA, exprB [, exprC] )
```

where:

- *indexVar* is a named scalar variable of type *integer* (usually named *i*, *j*, etc.); note that the *scope* of this variable is restricted to the implied-do loop, so it will not affect the value of the variable if used in other parts of the program
- *expr1*, *expr2*, ... are expressions (not necessarily of *integer* type), which may or may not depend on *indexVar*
- *exprA*, *exprB*, and *exprC* are scalar expressions (of *integer* type), denoting the lower bound, upper bound and (optional) increment step for *indexVar*

To illustrate the implied-do loops, we use them to re-write the operations above (for *weights* and *xRange*) in a more compact (but otherwise equivalent) form:

<sup>65</sup> Or surrounded by the pre-Fortran 2003 tokens `(/` and `/)`.

```

! index variable for implied-do still needs to be declared
integer :: i

xRange = [ (i, i=1,10) ] ! uses declaration above

real, dimension(0:8), parameter :: weights = &
[ 4./9., (1./9., 1./36., i=1,4) ]

```

The implied-do loop is eventually expanded, such that the list { `expr1`, `expr2`, ..., } is repeated for each value of the `indexVar`, using the appropriate value of the index variable for each repetition. For instance, in our second example above, the list { `1./9.`, `1./36.` } is repeated 4 times (and the value of the index variable is not used for computing any component).

### 2.6.4.6 Array Constructors for Multi-dimensional Arrays

So far, we only used array constructors for building 1D arrays. It is also possible, however, to construct multi-dimensional arrays, with a two-step procedure:

1. construct a 1D-array `tmpArray`
2. pass `tmpArray` to the intrinsic function `reshape`, to obtain a multi-dimensional array

In practice, the two steps are commonly combined into a single statement. The following example illustrates this, for constructing a  $10 \times 20$  matrix, where each element  $a_{i,j} = i * j$ :

```

real, dimension(10, 20) :: a = reshape( &
  source = [ ((i*j, i=1,10), j=1,20) ], &
  shape = [ 10, 20 ] &
)

```

where we also demonstrated the way in which implied-do loops can be *nested* (essentially, by replacing one or more of the expressions `expr1`, `expr2`, ..., discussed above by another implied-do loop).

In its basic form,<sup>66</sup> the `reshape` implicit function takes two arguments (denoted by optional keywords `source` and `shape`), both of them being 1D arrays, and where `shape` should be constant, and with non-negative elements.

The elements are read, in array element order, from the `source`-array, and written, also in array element order, to the result array.

## 2.6.5 I/O for Arrays

Just as we demonstrated in Sect. 2.4 for scalar variables, it is also essential to read/write (parts of) arrays from/to external devices. In principle, the same ideas could

<sup>66</sup> Additional arguments are supported, although not discussed here—see, e.g. Metcalf et al. [10].

be used, by simply treating individual array elements as scalar variables. However, there are several techniques related to array I/O which can simplify these operations. This section is devoted to these techniques.

### 2.6.5.1 Default Format (\*)

Just as for scalar variables, it is possible to let the system choose a default format, as in:

```

1 integer :: i, j ! dummy indices
2 integer, dimension(2,3) :: inArray = 0
3
4 write(*, '(a)') "Enter array (2x3 values):"
5 read(*, *) inArray
6 write(*, '(a)') "You entered:"
7 write(*, *) inArray

```

The input (provided for *line 5* in the listing above) can be provided over multiple records—the system will keep reading new records, until the elements in the I/O-list (whole array in our case) are satisfied.

The appearance of the output (generated by *line 7*) is, as in the case of scalars, platform-dependent. This was merely an aesthetic issue for scalars, but in the case of arrays it actually poses a serious problem, since the topological information of the array is effectively *lost*<sup>67</sup> (the lines in the output will not correspond, in most cases, to recognizable features of the array, such as rows and columns for 2D arrays). In the particular case of the previous listing the 6 array elements would normally fit on a single line of output.

In the remainder of this section, we discuss several methods for producing higher-quality output. Related to this, we also illustrate several methods for specifying the format specification, ranging from verbose to compact.

### 2.6.5.2 Implied-do Loops in the I/O-List

A first problem with the `write`-statement at *line 7* in the previous listing is that, when an array appears in the I/O-list, the I/O-system will effectively expand it internally to a list of array elements, taken in the array element order. We know, based on the discussion at the beginning of this section, that for a 2D array this order is the transpose of what would be needed to output the elements (given that Fortran I/O is record-based). This can be solved by modifying the I/O-list, so that it contains an implied-do loop instead of the array, as follows:

```

write(*, *) ( ( inArray(i,j), j=1,3), i=1,2 )

```

<sup>67</sup> Strictly speaking, it is still possible to deduce the coordinates of a specific element in the output list, by counting its position, and then comparing this with the expected array element order; however, this can hardly be called productive use of the programmer's time.

### 2.6.5.3 List of Formats (Verbose)

The previous listing causes the two rows of the array to be written on the same line. To separate them, we need to control the appearance of the output, using a customized format specifier, as we illustrated before for scalars. A first option to achieve this is to specify a verbose list of edit descriptors, as in:

```
write(*,'(x, i0, x, i0, x, i0, /, x, i0, x, i0, x, i0)' ) &
( ( inArray(i,j), j=1,3), i=1,2 )
```

### 2.6.5.4 Repeat Counts

The previous statement causes the two rows of the matrix to appear on separate lines, as intended. However, the format specifier is quite verbose, and it would be impractical to write in this form if the matrix were to be larger. We mentioned below that Fortran allows repeat counts to be placed in front of edit descriptors, or groups of edit descriptors within parentheses. In the current case, this can be used to make the format descriptor more compact, by factoring the `x, i0`-pattern:

```
write(*,'(3(x, i0), /, 3(x, i0))' ) &
( ( inArray(i,j), j=1,3), i=1,2)
```

### 2.6.5.5 Recycling of Edit Descriptors

Finally, we notice that Fortran has a mechanism for “recycling” edit descriptors, so that there can be more elements in the I/O-list than edit descriptors in the output format. When the I/O-subsystem “runs out” of edit descriptors, a new line of output is started, and the format specifier is re-used for the next elements in the I/O-list. This is perfect for our current purposes, as the output format can be further simplified using this feature:

```
write(*,'(3(x, i0))' ) &
( ( inArray(i,j), j=1,3), i=1,2)
```

## 2.6.6 Array Expressions

We emphasized above the usefulness of working with whole arrays and array sections, instead of manually iterating through the array elements with loops. Fortran allows a similar high level of abstraction for representing computations, with array

expressions. Specifically, most unary intrinsic functions and operators can take a whole array (or an array selection) as an argument, producing another array, with the same shape, through element-wise application of the operation. The same idea applies to binary operators, as long as the arguments are *conformable*. The following program uses these techniques to evaluate the functions  $\sin(x)$  and  $\sin(x) + \cos(x)/2$  on a regular grid, spanning the interval  $[-\pi, \pi]$ :

```

1  program array_expressions1
2  implicit none
3  integer, parameter :: N=100
4  real, parameter :: PI=3.1415
5  integer :: i
6  real, dimension(-N:N) :: &
7     xAxis = [ (i*(pi/N), i=-N,N) ], &
8     a = 0, b = 0
9
10
11  ! Compact array-expressions, using elemental functions.
12  a(i) = sin( xAxis(i) )
13  b(i) = sin( xAxis(i) ) + cos( xAxis(i) )/2.
14  b = sin(xAxis) + cos(xAxis)/2.
15
16  write(*, '(f8.4, 2x, f8.4, 2x, f8.4)') &
17     [ (xAxis(i), a(i), b(i), i=-N,N) ]
18 end program array_expressions1

```

Listing 2.29 src/Chapter2/array\_expressions1.f90

Note that the standard does not impose a specific order in which the elements of the result array for the expression are to be created. This allows compilers to apply hardware-specific optimizations (e.g. vectorization/parallelization). For this to be possible, all array expressions are completely evaluated, before the result is assigned to any variable. This makes array expressions behave differently from `do-loop` constructs which superficially seem equivalent to the array expression (so one needs to carefully examine any data dependencies between the different iterations of the `do-loops` when translating between the two forms of syntax). This was not the case for the two array expression examples above (*lines 12 and 14* in the listing), which could have also been written equivalently with a `do-loop` (although we recommend the previous, compact version):

```

do i=-N,N
  a(i) = sin( xAxis(i) )
  b(i) = sin( xAxis(i) ) + cos( xAxis(i) )/2.
enddo

```

However, the expression:

```

a(-(N-1):(N-1)) = ( a(-(N):(N-2)) + a(-(N-2):N) ) /2.

```

which assigns to each interior element of `a` an average value computed using its left and right neighbours, is **not** equivalent to the loop:

```

do i=-(N-1), (N-1)
  a(i) = ( a(i-1) + a(i+1) )/2.
enddo

```

We demonstrated above that some intrinsic functions (`sin`, `cos`, etc.) accept a scalar, as well as a whole array, as their argument.<sup>68</sup> Such functions are known in Fortran as `elemental`, and can also be defined by the programmer, for derived types, or for specific types of arrays. We provide a brief example for this, in Sect. 3.4.

### 2.6.7 Using Arrays for Flow-Control

Another array-oriented feature in modern Fortran consists of two specialized flow-control constructs. Just as the `if`, `case`, and `do` were demonstrated to produce more compact code when working with scalars, for arrays the `where` and `forall` constructs can be used to simplify array expressions, and to further avoid the need for manually expanding the expressions (with loops and element-based statements). As a general note, both of these constructs can be named and nested (see Metcalf et al. [10] for details).

#### 2.6.7.1 `where` Construct

The `where` construct can be used to restrict an array assignment only to elements which satisfy a given criterion. It is also known as *masked array assignment*. In many ways, it is the array-oriented equivalent of the `if`-construct, discussed for scalars. In its basic form, the syntax of `where` reads:

```
where( <logicalArray> )
  array1 = <array_expression1>
  array2 = <array_expression2>
  ..
end where
```

where `logicalArray`, `array1`, `array2`, etc., must have the same shape, and `logicalArray` may also be a logical expression (for example, comparing array elements to some scalar value).

For example, assume we have two arrays `a` and `b`, and that we want to copy inside `b` the `a`-values<sup>69</sup> that are lower than some scalar value `threshold`. This can be easily achieved with the `where` construct, as follows:

```
program where_construct1
  implicit none
  integer, parameter :: N = 7
  character(len=100) :: outFormat
  integer :: i, j
  real :: a(N,N) = 0, b(N,N) = 0, threshold = 0.5, &
         c(N,N) = 0, d(N,N) = 0 ! used in next examples

  ! write some values in a
  call random_number( a )
```

<sup>68</sup> Programmers familiar with C++ can think of this as a restricted form of function overloading.

<sup>69</sup> `random_number` is an intrinsic subroutine, described in Sect. 2.7.2.

```

! Create dynamic format, with internal-file(=string) outFormat.
! This way, the format is adjusted automatically if N changes.
write(outFormat, *) "(" , N, "(x, f8.2))"

write(*, '(a)') "a = "
write(*, fmt=outFormat) &
  ( (a(i,j), j=1,N), i=1,N )

! ** Masked array-assignment **
where( a > threshold )
  b = a
end where

write(*, '(/,a)') "b (after masked assignment) = "
write(*, fmt=outFormat) ( (b(i,j), j=1,N), i=1,N )
end program where_construct1

```

**Listing 2.30** `src/Chapter2/where_construct1.f90`

Similar to the `if`-construct, the `where`-construct could have been compacted, in this case, to a single line (since a single array assignment statement was present):

```
where( a > threshold ) b = a
```

Next, suppose we also want to copy over to array `c` the values of `a` that are smaller than half the threshold. We can extend the `where`-construct with an `elsewhere(logicalArray)` construct, similar to the `elseif`-branches we showed for `if`:

```

where( a > threshold )
  b = a
elsewhere( a < threshold/2. )
  c = a
end where

```

As a final extension of our example, let us assume that we want to copy over to array `d` the remaining values of `a`, which satisfy neither of the criteria (like the `else`-branch of `if`). This is achieved again with an `elsewhere`-branch, which does not have a `logicalArray` associated, as in:

```

where( a > threshold )
  b = a
elsewhere( a < threshold/2. )
  c = a
elsewhere
  d = a
end where

```

The logical arrays which define the masks (for the `where`- or `elsewhere`-branches) are first evaluated, and then the array assignments are performed in sequence, masked by the logical arrays (i.e. no assignment is performed for elements where the mask is `.false.`). This implies that, even if some assignments would alter the data used for evaluating the mask array,<sup>70</sup> such changes will not affect the remainder of the `where`-construct, for which the initially evaluated mask will be used.

<sup>70</sup> In our examples above, this would mean changing elements of `a`.

### 2.6.7.2 The `do concurrent` Construct

The `do concurrent` construct (introduced in Fortran 2008) can also be used for improving the performance and conciseness of array operations. Strictly speaking, the construct is more general, as it can also be used to work with scalar data. However, we discuss it here, as it is particularly useful for arrays, and also because it effectively supersedes another array-oriented construct (`forall`), which we do not cover in this text.<sup>71</sup>

We begin our brief discussion of this construct with a warning: as for many Fortran 2008 features, support for `do concurrent` was, at the time of writing, still incipient.<sup>72</sup>

The syntax of the construct is as follows:

```
do concurrent( [type_spec ::] list_of_indices_with_ranges &
[, scalar_mask_expression] )
  statement1
  statement2
end do
```

where `list_of_indices_with_ranges` can be an index range specification (as would appear after a normal `do`-loop), or a comma-separated list of such specifications (in which case, the construct is equivalent to a set of nested loops). We discuss the optional `type_spec` at the end of this section. The `scalar_mask_expression`, when present, is useful for restricting the statement application only to values of indices for which the expression evaluates to `.true.`. This is illustrated in the following example, where elements of matrix `a` which belong to a checkerboard pattern are copied to matrix `b`:

```
1 program do_concurrent_checkerboard_selection
2   implicit none
3   integer, parameter :: DOUBLE_REAL = selected_real_kind(15, 307)
4   integer, parameter :: N = 5 ! side-length of the matrices
5   integer :: i, j ! dummy-indices
6   real(kind=DOUBLE_REAL), dimension(N,N) :: a, b ! the matrices
7   character(len=100) :: outFormat
8
9   ! Create dynamic format, using internal file
10  write(outFormat, *)(" ", N, "(x, f8.2)")
11  ! Initialize matrix a to some random values
12  call random_number( a )
13
14  ! Pattern-selection with do concurrent
15  do concurrent( i=1:N, j=1:N, mod(i+j, 2)==1 )
16    b(i,j) = a(i,j)
17  end do
18
19  ! Print matrix b
```

<sup>71</sup> In many ways, `forall` is a more restricted version of `do concurrent`, which is why we prefer to describe only the latter. The syntax is very similar for both constructs. See, e.g. Metcalf et al. [10] for more details on `forall`.

<sup>72</sup> That being said, we found that both `gfortran` (version 4.7.2) and `ifort` (version 13.0.0) support this construct, with the exception of the type specification. Check the documentation of your compiler, for any flags that may need to be added to enable this feature (e.g. `-ftree-parallelize-loops=n`, with `n` being the number of parallel threads (for `gfortran`), or `-parallel` (for `ifort`))



```

20 write(*, '(/,a)') "b ="
21 write(*, fmt=outFormat) ( (b(i,j), j=1,N), i=1,N )
22 end program do_concurrent_checkerboard_selection

```

**Listing 2.31** src/Chapter2/do\_concurrent\_checkerboard\_selection.f90

Syntactically, the construct in *lines 15–17* in the previous listing could have been written using nested `do`-loops and an `if`, as in:

```

do i=1,N
  do j=1,N
    if( mod(i+j, 2)==1 ) then
      b(i,j) = a(i,j)
    end if
  end do
end do

```

so the version using `do concurrent` is obviously more compact. More importantly, the construct also enables some compiler optimizations with respect to the version using nested `do`-loops. There is a tradeoff, of course, because the restrictions on `do concurrent` do make it less general. Some of these (*restrictions*) are things that the compiler can check (and issue compile-time error if they are violated), while others cannot be checked automatically, and the programmer *guarantees* that they are satisfied.<sup>73</sup> For example:

- Most *restrictions* relate to preventing the programmer to branch outside the `do concurrent`-construct. Examples of mechanisms which can cause such branches are `return`, `go to`, `exit`, `cycle`, or `err=` (for error-handling). A safe rule of thumb is to avoid these statements.<sup>74</sup>
- Calling other procedures from the body of the construct is allowed, as long as these procedures are *pure*. This notion, discussed in more detail in the next chapter, implies that the procedure has *no side effects*; examples of side effects which would render procedures impure are:
  - altering program's state, in a global entity, or locally to the procedure, which may be used next time the procedure is called
  - producing output during one iteration, which is read during another iteration
- The programmer also *guarantees* to the compiler that there are no data dependencies between iterations (through shared variables, data allocated in one iteration and de-allocated in another iteration, or writing and reading data from an external channel in different iterations)

Given these limitations, using `do concurrent` may require some additional effort. However, for applications where performance is a priority, this is time

<sup>73</sup> Therefore, the program may successfully compile, but still contain bugs, if some of these implied guarantees do not actually hold!

<sup>74</sup> Strictly speaking, those which reference a labelled statement are allowed, as long as that statement is still within the `do concurrent`-construct.

well-spent, since it forces the programmer to re-structure the algorithms in ways which are favorable for parallelization at later stages (more about this in Sect. 5.3).

An interesting last note about this construct is that the standard also allows to specify the type of the indices within the construct (the type is always `integer`, but the `kind`-parameter can be customized). This is very convenient, since it brings type declarations closer to the point where the variables are used (otherwise these indices would need to be declared at the beginning of the (sub)program, as done in the previous example-program). For example, the pattern-selection portion in the earlier example-program could be written as:

```
do concurrent( integer :: l=1:N, m=1:N, mod(l+m, 2) == 1)
  b(l,m) = a(l,m)
end do
```

Note that, at the time of writing, most compilers still do not support this. However, it should be allowed in the near future.

### 2.6.8 Memory Allocation and Dynamic Arrays

In the examples so far, we only showed how to work with arrays whose shape is known at compile-time. This is often not the case in real applications, where this information may be the result of some computations, or may even be provided by the user at runtime. If this were a book about C++, now would definitively be the place to discuss pointers. In Fortran, however, this is not necessary<sup>75</sup> for dynamic-size arrays, which are supported through a simpler (and faster) mechanism, discussed in this section.

We often use the terms *static* and *dynamic* when discussing how memory is reserved for data entities. Generally speaking, memory for *static* objects is automatically managed by the OS. Examples of static entities are static global variables (defined through the `module`-facility, discussed later), variables local to a procedure, and procedure arguments (also covered later). Contrarily, *dynamic* objects require the programmer to explicitly make requests for acquiring and releasing regions of memory. Therefore, whereas for working with normal (static) arrays only a declaration is necessary, the workflow for dynamic arrays involves three steps:

1. **declaration:** Dynamic arrays are declared similarly to normal arrays. For example, a dynamic version of array `bigArray` (see Sect. 2.6.1) is given below:

```
integer, dimension(:,:,:), allocatable :: bigArray
```

<sup>75</sup> Pointers are still useful in many contexts, like for constructing more advanced data structures. They too are supported in Fortran, via the `pointer`-attribute (but Fortran pointers carry more information and restrictions than their C/C++ counterparts). We do not discuss this issue in this text—see, e.g. Metcalf et al. [10] or Chapman [1].

Note that there are two notable differences in the dynamic version:

- a. the *shape* of the array is not specified; instead, only the rank is declared (encoded as the number of `:`-characters in the list within the parentheses)
  - b. the `allocatable`-attribute needs to be added, to clarify that this is a dynamic array
2. **allocation:** Before working with array elements is allowed, memory has to be allocated, so that the exact shape of the array is specified. This is done with the `allocate`-statement, which has the form:

```
allocate( list_of_objects_with_shapes [, stat=statCode] )
```

where `statCode` is an (optional) integer scalar, set to zero by the system if the allocation was successful, or to some positive value if an error occurred (such as not enough memory to hold the arrays requested), and `list_of_objects_with_shapes` is a list of arrays, each followed by the explicit shape in parentheses (as would normally appear after the `dimension`-attribute if the arrays were static). For example, the following statements allocate the dynamic versions of arrays `xArray`, `bigArray`, and `zArray`, from Sect. 2.6.1:

```
integer :: statCode
allocate( xArray(16), bigArray(520,520,520), zArray(-100:100), stat=statCode )
```

After allocation, one can work with these arrays normally, as discussed before for the static case.

3. **deallocation:** A last concern related to dynamic arrays is to release the memory to the system, as soon as it is not needed by the program anymore. This is a highly recommended practice, both for performance reasons (because it reduces the amount of bookkeeping at runtime), and for increasing the readability of the programs (to signal the fact that the data is not used in other parts of the program). This step is achieved with the `deallocate`-function, which has the syntax:

```
deallocate( list_of_objects [, stat=statCode] )
```

where `statCode` has the same error-signalling role as before, and `list_of_objects` is a list of arrays. For example, the following statement releases the memory allocated above, for the arrays `xArray`, `bigArray`, and `zArray`:

```
deallocate( xArray, bigArray, zArray, stat=statCode )
```

Note that it is an error to attempt allocating an already-allocated array, or deallocating an already-deallocated (or never allocated) array. The allocation status of an

array may become difficult to track in larger programs, especially if the array is part of the global data and used by many procedures. The `allocated` intrinsic function can be used in such cases. For example:

```
allocated( xArray )
```

will return `.false.` before the `allocate`-call above, and after the `deallocate`-call; it will return `.true.`, however, between these two calls. Interestingly, since Fortran 2003, it is not necessary [13] to use this intrinsic function when we want to assign to the allocatable array another array (or array expression): in that case, allocation to the correct shape is automatically done by the Fortran runtime.

**Exercise 9** (*Array transversal order and performance*) Earlier in this chapter, we mentioned that array element order dictates the optimal array-transversal order for obtaining good performance. To test this, write a program which adds two cubic 3D-arrays (a and b), using nested `do`-loops. Measure the time required for the program to complete, for two different transversal scenarios:

```
do i=1,N
  do j=1,N
    do k=1,N
      a(i,j,k) = a(i,j,k) + b(i,j,k)
    enddo
  enddo
enddo

do k=1,N
  do j=1,N
    do i=1,N
      a(i,j,k) = a(i,j,k) + b(i,j,k)
    enddo
  enddo
enddo
```

### Hints:

- The length of the cube's side (N) should be large enough to be representative for a real-world scenarios (i.e. the whole arrays should not fit in the cache). For example, take  $N = 813$ , and 32bit real array elements. It is easier to use allocatable arrays.<sup>a</sup>
- To improve the accuracy of the result, wrap the code above within another loop, so that the operations are performed, say,  $N_{\text{repetitions}} = 30$  times.<sup>b</sup>
- It is also instructive to test the programs with several compilers, because some highly-optimizing compilers (like `ifort`) may recognize performance “bugs” like these in simple programs, and correct the problem

internally (but this can fail in more complex scenarios, so learning about these issues is still valuable). Also, compilers can simply “optimize away” code when the computation results are not used, so try to print some elements of `a` at the end of the computation.

<sup>a</sup>Most systems have some limits for the size of static data (“stack size”). Therefore, large static arrays would require adjusting these limits and, possibly, adjusting the “memory model” through compiler flags.

<sup>b</sup>This reduces the effect of system noise, and it also provides a “poor man’s” solution for reducing the relative importance of the (de)allocation overhead—a more accurate approach is to benchmark the computational parts exclusively, using techniques discussed later, in Sect. 2.7.

## 2.7 More Intrinsic Procedures

In the course of our discussion so far, we have already mentioned some of the many intrinsic procedures offered by Fortran. In this section, we describe a few additional ones, which would not easily fit into the previous sections, but are nonetheless common practice. We discuss later (in Chap. 3) how to define custom procedures.

### 2.7.1 Acquiring Date and Time Information

Some ESS applications need to be concerned with the current date and time. The `date_and_time` intrinsic subroutine is appropriate for this. When calling this, one can pass (as an argument) an `integer`-array, of size 8 or more. The Fortran-runtime will then fill the components with `integer`-values, as described in Table 2.3.

A very common application is timing a certain portion of code, as a quick way for profiling parts of a program. In principle, using `date_and_time` before and after the part of the algorithm to be profiled could be used, but this limits the time

**Table 2.3** Data inserted into components of `curr_date_and_time`

| Component # | Meaning                              | Component # | Meaning      |
|-------------|--------------------------------------|-------------|--------------|
| 1           | Year                                 | 5           | Hour         |
| 2           | Month                                | 6           | Minutes      |
| 3           | Day                                  | 7           | Seconds      |
| 4           | Time difference (minutes) w.r.t. GMT | 8           | Milliseconds |

resolution that can be achieved. Fortran also has the `cpu_time` intrinsic for such purposes, which provides microsecond precision on many platforms.

A complete program, demonstrating these functions, is given below:

```

program working_with_date_and_time
  implicit none
  ! for date_and_time-call
  integer :: dateAndTimeArray(8)
  ! for cpu_time-call
  real :: timeStart, timeEnd
  ! variables for expensive loop
  integer :: mySum=0, i

  call date_and_time(values=dateAndTimeArray)
  print*, "dateAndTimeArray =", dateAndTimeArray

  call cpu_time(time=timeStart)
  ! expensive loop
  do i=1, 1000000000
    mySum = mySum + mySum/i
  end do
  call cpu_time(time=timeEnd)
  print*, "Time for expensive loop =", timeEnd-timeStart, "seconds", &
    " ", mySum = ", mySum
end program working_with_date_and_time

```

**Listing 2.32** `src/Chapter2/working_with_date_and_time.f90`

Some precautions apply to uses of `cpu_time`:

- results are generally non-portable (since the resolution is not standardized, to allow higher precision for platforms which support it)
- even if no other demanding programs seem to be running on the system, the timing results will fluctuate, due to ever-present “system noise” (the OS needs to continuously run some internal programs, to maintain proper operation)
- the function is not useful for parallel applications; for example, in a parallel program using OpenMP, the `omp_get_wtime`-subroutine should be used instead
- while convenient for quick tests, this approach to profiling does not scale (just as `print`-based debugging does not scale well for complex bugs); many manufacturers, as well as open-source projects, offer much more convenient tools for complex scenarios.

## 2.7.2 Random Number Generators (RNGs)

Statistical methods form the basis of many powerful algorithms in ESS. For example, stochastic parameterizations are commonly used in models, to simulate the effects of processes at smaller spatial scales (clouds, convection, etc.), which are not resolved by the (usually severely coarsened) model mesh. A basic necessity for many such algorithms is the ability to generate sequences of random numbers. This may seem

a simple technicality but, in fact, it invites a philosophical question, since computer algorithms are supposed to produce deterministic outcomes.<sup>76</sup>

Nonetheless, many algorithms can produce sequences which are often “sufficiently random”, despite being deterministic. Fortran implementations also provide such algorithms, via the `random_number` intrinsic subroutine. The following program uses it to estimate  $\pi$ .

```

7  program rng_estimate_pi
8  implicit none
9  integer, parameter :: NUM_DRAWS_TOTAL=1e7
10 integer :: countDrawsInCircle=0, i
11 real :: randomPosition(2)
12 integer :: seedArray(16)
13
14 ! quick method to fill seedArray
15 call date_and_time(values=seedArray(1:8))
16 call date_and_time(values=seedArray(9:16))
17 print*, seedArray
18 ! seed the RNG
19 call random_seed(put=seedArray)
20
21 do i=1, NUM_DRAWS_TOTAL
22   call random_number( randomPosition )
23   if( (randomPosition(1)**2 + randomPosition(2)**2) < 1.0 ) then
24     countDrawsInCircle = countDrawsInCircle + 1
25   end if
26 end do
27 print*, "estimated pi =", &
28 4.0*( real(countDrawsInCircle) / real(NUM_DRAWS_TOTAL))
29 end program rng_estimate_pi

```

**Listing 2.33** `src/Chapter2/rng_estimate_pi.f90`

Note that we used another intrinsic subroutine (`random_seed`), to compensate for the deterministic nature of the *random number generator* (RNG).<sup>77</sup> To link with the previous discussion, we use two calls to the `date_and_time` intrinsic, to obtain a seed array.<sup>78</sup> This is not an “industrial-strength” solution, since the date information is not completely random (and some components like the time zone are in fact constant).<sup>79</sup>

The algorithm itself is based on placing random points within a square  $2D$ -domain, and checking what fraction of those fall within the largest quarter-of-circle inscribed in the square. This is a classical example of what is known as the Monte-Carlo approach to simulation.

<sup>76</sup> This is fundamentally different from randomness in the physical sense, which is driven by the quantum-probabilistic processes at the atomic scale. These effects are then amplified at the mesoscopic scales, due to the large number of degrees of freedom of the system (e.g. climate system, see Hasselmann [6]).

<sup>77</sup> In situations where perfect reproducibility of results is necessary, the seeding step could be skipped. However, a more scientifically-robust method to achieve this is to use a sequence of random numbers large enough that the reproducibility is achieved algorithmically.

<sup>78</sup> You can check how large the array needs to be for your platform, by calling the seed function like `call random_seed(size=seedSize)`, where `seedSize` is an integer scalar, inside which the result of the inquiry will be placed.

<sup>79</sup> A better solution for seeding may be to use the entropy pool of the OS. In Linux, you can read data from the file `/dev/random` (see, e.g. Exercise 10).

**Exercise 10** (*Accurate  $\pi$* ) Modify the previous program, so that it reliably recovers the first 7 digits after the decimal dot of  $\pi$ .

**Hints:** you will need to ensure that the variables involved have a `kind` which is accurate enough. Also, to rule out “accidental” convergence, it is a good idea to check that the convergence criterion remains satisfied for several (say, 100) Monte-Carlo draws in a row.

As a final note on this topic, for scientific applications it is often important to ensure the RNG passes certain quality criteria—usually a batch of tests. Thus, a hierarchy of RNG-algorithms exists, relative to which the `random_number` intrinsic may not be the most suitable. For an in-depth discussion of this topic, please refer to Press et al. [12].

## References

1. Chapman, S.J.: Fortran 95/2003 for Scientists and Engineers. McGraw-Hill Science/Engineering/Math, New York (2007)
2. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT Press, Cambridge (2009)
3. Fraedrich, K., Jansen, H., Kirk, E., Luksch, U., Lunkeit, F.: The planet simulator: towards a user friendly model. *Meteorol. Z.* **14**(3), 299–304 (2005)
4. Goldberg, D.: What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.* **23**(1), 5–48 (1991)
5. Hager, G., Wellein, G.: Introduction to High Performance Computing for Scientists and Engineers. CRC Press, Boca Raton (2010)
6. Hasselmann, K.: Stochastic climate models Part I. *Theory Tellus* **28A**(6), 473–485 (1976)
7. Kirk, E., Fraedrich, K., Lunkeit, F., Ulmen, C.: The planet simulator: a coupled system of climate modules with real time visualization. Technical Report 45(7), Linköping University (2009)
8. Marshall, J., Plumb, R.A.: Atmosphere, Ocean and Climate Dynamics: An Introductory Text, 1st edn. Academic Press, Boston (2007)
9. Mehlhorn, K., Sanders, P.: Algorithms and Data Structures: The Basic Toolbox. Springer, Berlin (2010)
10. Metcalf, M., Reid, J., Cohen, M.: Modern Fortran Explained. Oxford University Press, Oxford (2011)
11. Overton, M.L.: Numerical Computing with IEEE Floating Point Arithmetic. Society for Industrial and Applied Mathematics, Philadelphia (2001)
12. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: The art of parallel scientific computing. Numerical Recipes in Fortran 90, vol. 2, 2nd edn. Cambridge University Press, Cambridge (1996). also available as <http://apps.nrbook.com/fortran/index.html>
13. Reid, J.: The new features of Fortran 2003. *ACM SIGPLAN Fortran Forum* **26**(1), 10–33 (2007)



Introduction to Modern Fortran for the Earth System  
Sciences

Chirila, D.B.; Lohmann, G.

2015, XXII, 250 p. 15 illus., 10 illus. in color., Hardcover

ISBN: 978-3-642-37008-3