Old programming idioms explained

Arjen Markus

May 31, 2023

1 Introduction

In the more than 60 years of its existence up to now the Fortran programming language has undergone many changes, both in accordance with general insights in programming paradigms and in reaction to developments in computer hardware. This document discusses some of the idioms one finds in old FORTRAN packages and their modern alternatives.

Please be aware that the use of lowercase and uppercase forms of the name is not a whimsicality but rather marks a revolution within the language that was started with the publication of the Fortran 90 standard in 1990(?). Throughout this document we will use this distinction which is much more than merely typographic.

The set-up is simple and ad hoc: we discuss various idioms that have been used in the past decades and present contemporary equivalents or alternatives. Attempts are made to present them in a systematic way, but that mostly means grouping related topics.

2 Floating-point numbers

Nowadays, most computers use the IEEE format for representing floating-point numbers. The two main types you will encounter are single-precision reals, occupying four bytes of memory, and double-precision reals, occupying eight bytes. Fortran of old, including FORTRAN, favours single-precision – without any decoration a literal number like `1.23456789` is single-precision, whereas many other languages use double-precision reals by default.

Back in the days before the IEEE standard reals [**?**] were represented in many different ways and also the arithmetic operations we normally take for granted were not fully portable. This led to all manner of complications if you wanted to make your program portable, and that was and is certainly a goal of Fortran.

2.1 REAL*4 and REAL*8

One of the many extensions that were added by compiler vendors was the use of an asterisk (*) to indicate the precision:

```
*
*     Single precision real
*
      real*4 x
*
*     Double precision real
*
      real*8 y
*
*     Single precision complex
*
      complex*8 z
```

The number indicates the number of bytes that a real would occupy. This has never been part of a FORTRAN or Fortran standard. The *kind* feature in Fortran is much more flexible, as it can capture aspects of the representation of floating-point numbers beyond mere storage.[1]

> *Note:* I have used a Convex computer in the distant past that actually had two different types of floating-point numbers that both were single-precision. One was structured according to the IEEE standard and the other was a native format. The difference for all intents and purposes was the interpretation of the exponent. The native format was said to be a bit faster, but they occupied the same storage, four bytes.
>
> For the same bit pattern the *values* differed by a factor 4.

2.2 Literals in the source code

One thing to keep in mind: if a literal number occurs in the source code, it is interpreted as it appears, independent of the context. For Fortran this has been standardised: an expression on the right-hand side is evaluated independently of the left-hand side. More concretely:

```
double precision pi = 3.14159265358979323846264338327950288419716939937510
```

may look to specify π in some 50 decimals, but to the compiler it is merely a slightly bizarre way of expressing it in *single-precision*, so actually only six or seven significant decimals. To get *double-precision*, you need to add a *kind* or, as it was in FORTRAN, a "d" exponent (with some excess decimals removed):

[1] The current standard defines a general model for representing real numbers. This encompasses the IEEE formats, but is in fact more general.

```
    double precision pi = 3.14159265358979323846264 3d0
```

You can see the difference if you run this program:

```
program diff_double_precision
    implicit none

    double precision, parameter :: pi_1 = 3.14159265358979323846264 3
    double precision, parameter :: pi_2 = 3.14159265358979323846264 3d0

    write(*,*) 'Difference: ', pi_1 - pi_2
end program diff_double_precision
```

which prints (you may expect slight differences in the last few decimals with different compilers):

```
 Difference:    8.7422780126189537E-008
```

Some old FORTRAN compilers seem to have been less strict about the dichotomy between the left-hand side and the right-hand side and would indeed interpret such literal numbers as double-precision.

Another thing to keep in mind is that many compilers, both new and old, allow for compiler options that turn the *default* precision for a variable declared as `real` into *double precision*. If a program relies on this behaviour, then you need to carefully check the code.

2.3 Input in the absence of a decimal point

Disk storage nowadays is all but endless, but this luxury did not exist in the old days. This may have been the reason for a little known or used feature in input of real numbers: if a string representing a real number does not contain a decimal point, then the *input format* may insert it.

Here is an example, using internal I/O to make it self-contained (see also the file `input_no_point.f90`):

```
program show_insert_point
    implicit none

    real :: x
    character(len=10) :: string

    string = '1234'

    read( string, '(f4.0)' ) x
    write(*,*) x

    read( string, '(f4.2)' ) x
    write(*,*) x
end program show_insert_point
```

It produces:

```
1234.00000
12.3400002
```

With the format in the second read statement a decimal point is inserted!

It may have been useful in the past but it does suggest that to avoid surprises, you better not use input format with a prescribed number of decimals.

3 Control structures

Placeholder:

- Nested do-loop
- Simulated do-while
- Jumping out of a do-loop
- IF-constructions, including "select-case"

4 Memory management

In the decades leading up to the FORTRAN 77 standard, memory management was simple: declare what memory you need via statically sized arrays and that is it. There was no dynamic memory allocation, at least not in the FORTRAN language. It may surprise you, but even the concept of an operating system that took care of the computer was fairly new, as illustrated by a 1971 book by D.W. Barron, titled "Computer Operating Systems". Quoting from the book's jacket:

> As the operating system is becoming an important part of the software complex accompanying a computer system. A large amount of knowledge about the subject now exists, mainly in the form of papers in computer journals. It is thus time for a book that coordinates what is known about operating systems.

There are, however, a few aspects of FORTRAN that make the story a bit more complicated: `COMMON` blocks, `EQUIVALENCE` and the `SAVE` statement. All three will be discussed here.

4.1 COMMON blocks

Variables, be they scalars or arrays, are normally passed via argument lists between program units (the main program, subroutines or functions). This is the immediately visible part. But you can also pass variables via `COMMON` blocks. These constitute a form of global *memory*, but not of global *variables*, as a `COMMON` block merely allocates memory and the mapping of memory locations onto variables is up to the program units themselves. For instance:

```
SUBROUTINE SUB1
COMMON /ABC/ X(10)
...
END

SUBROUTINE SUB2
COMMON /ABC/ A(5), B(5)
...
END
```

The `COMMON` block `/ABC/` appears in two subroutines, but in subroutine `SUB1` it is associated with the array `X` of 10 elements and in the subroutine `SUB2` it is associated with two arrays, `A` and `B`, both having five elements. The memory is shared, so that if you set `X(1)` to, say, `1.1` in subroutine `SUB1`, then on the next call to subroutine `SUB2`, the array element `A(1)` will have that same value, as they occupy the same memory location.

`COMMON` blocks should have the same size in all locations in the program's code where they occur. That is difficult to ensure, hence it was common (no pun intended) to put the declaration of `COMMON` blocks in so-called include files. Each program unit that needed to address the memory allocated via these `COMMON` blocks could then use the `INCLUDE` statement to have the compiler insert the literal text of that include file.[2]

There are various ways that `COMMON` blocks were used:

- Variables in `COMMON` blocks are persistent. At least, that was a very common occurrence. The rules in the FORTRAN standard are more complicated, but certainly with the `SAVE` statement you can rely on these variables to retain the values between calls to a routine.

- Often routines in a library have to cooperate: one routine is used to set options and other routines do the actual work. By using one or more `COMMON` blocks these options do not need to be passed around via the argument list.

- Together with `EQUIVALENCE` statements you could use the `COMMON` blocks to share workspace. Remember: back in the days memory was much and much more precious and scarce than it is now. So, defining work arrays `WORK` (of type real) and `IWORK` (of type integer) and making them equivalent to each other, you could save on memory, if these arrays are not used at the same time.

Nowadays, it is much easier to pass large amounts of essentially private data around, simply define a suitable derived type. Also, it is easy to allocate work arrays as you require them and release them again when done.

A special `COMMON` block was the so-called *blank* `COMMON` block. It had no name and it did not have to be declared with the same size in all parts of the

[2]The INCLUDE statement was actually a common compiler extension.

program. In fact, on some systems it could be used as a flexible reservoir of memory, in much the same way as you have the heap nowadays. But this facility was an extension to the standard.

4.2 The SAVE statement

According to the FORTRAN standard a local variable in a function or subroutine does not retain its value between calls, unless it has the `SAVE` attribute:

```
      SUBROUTINE ACCUM( ADD )
*
*     Accumulate the counts
*
      INTEGER ADD

      INTEGER TOTAL
      SAVE    TOTAL
      DATA    TOTAL / 0 /

      TOTAL = TOTAL + ADD
      IF ( TOTAL > 100 ) THEN
          WRITE(*,*) 'Reached: ', TOTAL
      ENDIF
      END
```

However, some implementations, notably on DOS/Windows, used static storage for these local variables, which meant that the variables would *seemingly* retain their values, even without the `SAVE` statement. If a program relied on this property and was ported to a different environment, all manner of havoc could be raised.

> *Note:* I have actually had lively, but not necessarily pleasant, debates on whether the behaviour either way was correct. Sometimes the unexpected behaviour was claimed to indicate a compiler bug.

4.3 The initial values of (local) variables

A feature related in a way to the `SAVE` statement is the fact that in both FORTRAN and Fortran variables do not get a particular initial value, unless they have the `SAVE` attribute, implicitly or explicitly. With older compilers local variables may be stored in static memory and quite often they may have an initial value of zero or whatever the equivalent is for the variable's type, but that is in all cases simply a random circumstance. *Never assume that a variable that has not been explicity given a value, has a particular value.*

You can set the initial value in FORTRAN via the `DATA` statement:

```
      LOGICAL FIRST
      DATA FIRST / .TRUE. /
```

This means that at the first call to the subroutine holding this variable `FIRST`, it has the value `.TRUE.`. You can later set it to `.FALSE.` to indicate that the subroutine has been called at least once before, so that no initialisation is needed anymore:

```
* Subroutine that sums the values we pass
      SUBROUTINE SUM( x )
      INTEGER X

      INTEGER TOTAL
      LOGICAL FIRST
      DATA FIRST / .TRUE. /

      IF ( FIRST ) THEN
          FIRST = .FALSE.
          TOTAL = 0
      ENDIF

      TOTAL = TOTAL + X
      END
```

(JUst a variation on the previous example). This is not a very interesting routine, but it illustrates a typical use.

To emphasize: This type of initialisation is done so that the variables in question have the designated value at the first call. If you change the value, then they retain that new value. No reinitialisation occurs. (Actually, the value is not set on the first call, but rather is part of the data section of the program as a whole. There is no separate assignment.)

The `DATA` statement is not executable, it normally appears somewhere in the section that defines the variables, but it may occur elsewhere – most FORTRAN and Fortran compilers are not strict about it.

There is some peculiar syntax involved:

```
REAL X(100)
DATA X / 1.0, 98*0.0, 100.0 /
```

You can repeat values in much the same way as with edit descriptors in format statements: a count followed by an asterisk ("*") and the value to be repeated. It is also possible to use implied do-loops:

```
INTEGER I
REAL X(100)

DATA (X(I), I = 1,100) / 100*1.0 /
```

While it is more usual to set the values together with the declaration of a variable nowadays, like:

```
integer :: i
real    :: x(100) = [ (1.0, i = 1,size(x)) ]
```

the DATA statement is more versatile, because it is not necessary to set the values for an array in one single statement:

```
integer :: i
real    :: x(100)

data (x(i), i = 1,50)   / 50*1.0 /
data (x(i), i = 51,100) / 50*0.0 /
```

So, this old-fashioned statement may have its uses still.

Another peculiarity: the DATA statement has effect on the size of the object file:

TODO

4.4 Initialising variables in COMMON blocks: BLOCK DATA

The DATA statement plays an important role when it comes to initialising variables in a COMMON block. Since the COMMON blocks may appear in more than one subprogram (main program, subroutines, functions), they cannot be initialised in the same way as ordinary variables: which DATA statement should prevail, if several initialise the same COMMON variables?

Thus enter the BLOCK DATA program unit!

It is the only way to initialise variables in a COMMON block and it is special, because it is not executable. The peculiar consequence is that you cannot put it in a library: there is no reference to it, unlike with subroutines and functions, so it would never be loaded. Instead, you will normally put in the same file as the main program or link against its object file explicitly.

The general layout is:

```
BLOCK DATA
... COMMON blocks ...
... DATA statements ...
END
```

Here is a small example of this effect:[3]

```
gfortran -c block.f90
gfortran -o common1 common.f90 block.f90
gfortran -o common2 common.f90
```

The block.f90 source file is:

[3] I use the gfortran compiler to illustrate such effects, but it would be similar with other compilers. And since most if not all FORTRAN features are still supported in Fortran, I also use free-form sources.

```
BLOCK DATA
COMMON /ABC/ X
DATA X /42/
END
```

The `common.f90` source file is:

```
PROGRAM PRINTX
COMMON /ABC/ X
WRITE(*,*) 'Expected value of X = 42:'
WRITE(*,*) 'X = ', X
END
```

Program `common1` prints the value 42, whereas the other program prints 0. If the `BLOCK DATA` program unit had been an ordinary program unit, the building of this version would have failed on an unresolved symbol or the like.

4.5 Work arrays

In the old days you could encounter arguments to a routine that represented such workspace. Usually you would have to declare the arrays to a size that matches the problem at hand. Here is an example from the *LAPACK* library for linear algebra:

```
      SUBROUTINE DGELS( TRANS, M, N, NRHS, A, LDA, B, LDB, WORK, LWORK,
     $                  INFO )
*
*  -- LAPACK driver routine (version 3.2) --
*  -- LAPACK is a software package provided by Univ. of Tennessee,    --
*  -- Univ. of California Berkeley, Univ. of Colorado Denver and NAG Ltd..--
*     November 2006
*
*     .. Scalar Arguments ..
      CHARACTER          TRANS
      INTEGER            INFO, LDA, LDB, LWORK, M, N, NRHS
*     ..
*     .. Array Arguments ..
      DOUBLE PRECISION   A( LDA, * ), B( LDB, * ), WORK( * )
*     ..
```

In this case, the argument `WORK` is a double-precision array of size `LWORK`. In the comments that document the use of this routine the precise usage is described:

```
*  WORK    (workspace/output) DOUBLE PRECISION array, dimension (MAX(1,LWORK))
*          On exit, if INFO = 0, WORK(1) returns the optimal LWORK.
*
*  LWORK   (input) INTEGER
```

```
*          The dimension of the array WORK.
*          LWORK >= max( 1, MN + max( MN, NRHS ) ).
*          For optimal performance,
*          LWORK >= max( 1, MN + max( MN, NRHS )*NB ).
*          where MN = min(M,N) and NB is the optimum block size.
*
*          If LWORK = -1, then a workspace query is assumed; the routine
*          only calculates the optimal size of the WORK array, returns
*          this value as the first entry of the WORK array, and no error
*          message related to LWORK is issued by XERBLA.
```

This means that for a particular case you can either use one of the formulae or the special value `-1` for `LWORK` to obtain an optimal value. The work array itself would still be a statically declared array.

Note that with the current features of Fortran the interface could be greatly simplified:[4]

```
      SUBROUTINE DGELS( TRANS, A, B, INFO )
*
*     .. Scalar Arguments ..
      CHARACTER, INTENT(IN) :: TRANS
      INTEGER, INTENT(OUT)  :: INFO
*     ..
*     .. Array Arguments ..
      DOUBLE PRECISION, INTENT(INOUT) ::  A(:,:), B(:,:)
*     ..
```

provided the interface is made explicit via a module or an interface block.

5 Subjects

- array(*) versus array(:)
- array(10) as the starting point
- COMMON blocks and BLOCK DATA, named and blank COMMON
- history of computers:
  - hardware
  - memory management
  - tools like source code control systems
  - connections between computers, Internet
- equivalence
- constants as actual arguments
- intent
- temporary arrays - non-contiguous arrays
- implicit types
- double precision versus kind

[4] Intentionally left in fixed form.

- checking interfaces
- separate compilations, the misunderstanding of one routine per file
- fixed form and spaces
- standard input and output
- LU-numbers 5 and 6 (and 7)
- command-line arguments for file names
- real do-variables
- entry
- SAVE-attribute and SAVE as compiler property
- ASSIGN
- arithmetic IF
- computed GOTO
- F66: one-pass do-loops
- statement functions
- six characters
- numerical binary representations versus IEEE (IBM, Cray, Convex)
- big-endian and little-endian
- double complex
- unformatted versus binary files
- list-directed input and output - also: /
- narrow formats
- input formats with decimals - F5.1 and there is no decimal point in the input
- double precision :: x = 1.01 not the same as double precision :: x = 1.01d0
- use of d00 in input