Old programming idioms explained

Arjen Markus

June 3, 2023

1 Introduction

In the more than 60 years of its existence 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 was widely adopted [**?**], 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:* Such notation is sometimes used for integers and logicals as well. Again the *kind* feature is much more useful than merely indicating the storage size.
>
> *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
```

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

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):

```
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"
- ASSIGN statement
- Computed GOTO
- Three-way IF

FORTRAN came with a small number of control constructs and it was quite usual to construct other control flows via `IF` and `GOTO` statements. It inherited some constructs from its predecessors that are very uncommon nowadays: the three=way `IF` and the computed `GOTO`, as well as the `ASSIGN` statement. This part of the document highlights these ancient idioms.

3.1 Ordinary and nested DO-loops

The ordinary `DO` loop in FORTRAN looks like this:

```
      DO 110 I = 1,10
          ... do something useful ...
  110 CONTINUE
```

The statement label `110` indicates the end of the `DO` loop and anything in between is repeatedly executed. The Fortran equivalent is, unsurprisingly:

```
do i = 1,10
    ... do something useful ...
enddo
```

But there are a few more things to say about these `DO` loops. First of all, the statement label needs not appear with a `CONTINUE` statement. It could very well be put on the last executable statement:

```
      SUM = 0.0
      DO 110 I = 1,10
  110     SUM = SUM + ARRAY(I)
```

It can even be used for multiple, nested, `DO` loops:

```
      SUM = 0.0
      DO 110 J = 1,10
      DO 110 I = 1,10
  110     SUM = SUM + ARRAY(I,J)
```

To skip a part of the calculation, you can use a `GOTO` statement, where in Fortran you would use a `cycle` or `exit` statement:

```
*
* Sum the positive elements only and only if the sum
* remains smaller than 1.0
*
      SUM = 0.0
      DO 110 I = 1,10
          IF ( ARRAY(I) .LE. 0.0 ) GOTO 110
          IF ( SUM .GT. 1.0 ) GOTO 120
  110     SUM = SUM + ARRAY(I)
  120 CONTINUE
```

The example is a little contrived, so that you can see the use of the `GOTO` statement for both `cycle` and `exit`. The modern equivalent becomes:

```
!
! Sum the positive elements only and only if the sum
! remains smaller than 1.0
!
      SUM = 0.0
      DO I = 1,10
          IF ( ARRAY(I) <= 0.0 ) CYCLE
          IF ( SUM > 1.0 ) EXIT
          SUM = SUM + ARRAY(I)
      ENDDO
```

Note that sharing statement labels in a nested `DO` loop makes it difficult to see what a statement `GOTO endlabel` should mean: skip an iteration or skip the rest of the inner `DO` loop:

```
      SUM = 0.0
      DO 110 J = 1,10
      DO 110 I = 1,10
          IF ( SUM .GT. 1.0 ) GOTO 110
  110     SUM = SUM + ARRAY(I,J)
```

In FORTRAN 66 (also known as FORTRAN IV) there was a significant difference with the `DO` loop you find in current Fortran: a `DO` loop would always be run at least once! This is due to the location of the check on the iteration condition, whether it is put at the start or at the end of the `DO` loop. Many compilers still provide an option to allow for the FORTRAN 66 semantics,[**?**] which includes this feature:[2]

```
* With the right compiler options, print this line once!
      DO 110 I = 1,0
          WRITE(*,*) 'FORTRAN 66: ', I
  110 CONTINUE
      WRITE(*,*) 'Current value of i:', i
```

With FORTRAN 66 semantics the sample program (see `f66_loop.f90`) prints:

```
 FORTRAN 66:            1
 Current value of i:           2
```

With modern semantics it prints:

```
 Current value of i:           1
```

This can result in subtle but nasty differences, if you are unaware of what was meant!

3.2 Simulating a DO-WHILE loop

There was no explicit `DO WHILE` construct in FORTRAN, at least not in the standard. Therefore you would need to simulate it using any of the following methods:

A `DO` *loop with a large upper bound:*

```
* Find the right line in a file
      DO 110 I = 1,10000000
          READ( 10, '(A)' ) LINE
          IF ( LINE(1:1) .NE. '*' ) THEN
              GOTO 120
          ENDIF
  110 CONTINUE
```

[2] I could not find such a flag for the gfortran compiler, but for Intel Fortran oneAPI it is -f66.

```
  120 CONTINUE
* Found the start of the information, proceed
      ...
```

A combination of statement labels and `GOTO` *– check at the start:*

```
* Find the right line in a file
      READ( 10, '(A)' ) LINE
  110 CONTINUE
      IF ( LINE(1:1) .NE. '*' ) GOTO 120
      READ( 10, '(A)' ) LINE
      GOTO 110
  120 CONTINUE

* Found the start of the information, proceed
      ...
```

(This example is a bit artificial to keep it in line with the other two, but similar constructs with different processing definitely occur in practice!)

A combination of statement labels and `GOTO` *– check at the end:*

```
* Find the right line in a file
  110 CONTINUE
      READ( 10, '(A)' ) LINE
      IF ( LINE(1:1) .EQ. '*' ) GOTO 110

* Found the start of the information, proceed
      ...
```

A modern equivalent would either use the `DO WHILE` loop or the unlimited `DO` loop:

```
!
! Find the right line in a file
!
read( 10, '(a)' ) line
do while (line(1:1) == '*' )
    read( 10, '(a)' ) line
enddo

!
! Found the start of the information, proceed
!
...
```

Or:

```
!
! Find the right line in a file
!
do
    read( 10, '(a)' ) line
    if (line(1:1) == '*' ) exit
enddo

!
! Found the start of the information, proceed
!
...
```

The precise location of the check on the condition depends on what the purpose is and whether you can actually check it at the start of the loop, as with a `DO WHILE`, or whether you require some preliminary calculation first. If you want to convert old-style source code, beware that the logic may sometimes have to be reverted, particularly if the condition comes at the end of the loop.

Note for self: https://www.intel.com/content/www/us/en/docs/fortran-compiler/developer-guide-reference/2023-0/f66.html

3.3 Three-way IF statements and computed GOTOs

Two types of statements that are quite alien to what you find in modern-day programming languages are the three-way or artihmetic `IF` statement and the computed `GOTO` statement. The latter could be used to simulate a `select case` construct, the first on the other hand was, in modern eyes, an unusual predecessor of the `IF ... ELSE ... ENDIF` block.

A *computed* `GOTO` *statement* takes a list of statement labels and a single integer expression:

```
      GOTO (100, 200, 300) JMP
  100 CONTINUE
      WRITE(*,*) 'Jump: 1'
      GOTO 400
  200 CONTINUE
      WRITE(*,*) 'Jump: 2'
      GOTO 400
  300 CONTINUE
      WRITE(*,*) 'Jump: 3'
  400 CONTINUE
      ... the rest ...
```

Depending on the value of this expression (the value of `JMP` in the above example, the control would jump to the Nth label. If the value was zero or lower, the `GOTO` would not be executed and the program control would simply continue with the

next statement. This is the case too with a value that is larger than the number of statement labels. (See as an illustration the source file `computed_goto.f90`)

Since there is nothing special about the statement labels the control would jump to, you had to make sure to jump somewhere else after the handling of each case. In the example that is done by jumping to label 400.

The `select case` construct of Fortran is better behaved, as you do not have to take care of jumping to the end yourself and it is possible to select the case via strings as well as integer values or even ranges.

There is nothing particularly magic about the *three-way* `IF` *statement*. But you need to know how it works:

```
      IF (IVALUE) 100, 200, 300
  100 CONTINUE
      WRITE(*,*) 'Value is negative - ', IVALUE
      GOTO 400
  200 CONTINUE
      WRITE(*,*) 'Value is zero - ', IVALUE
      GOTO 400
  300 CONTINUE
      WRITE(*,*) 'Value is positive - ', IVALUE
  400 CONTINUE
      ... the rest ...
```

The action, a jump to one of the three statement labels, to be taken depends on the *sign* of the integer expression. Often two of the statement labels would be the same, as two possibilities are more common than three. To see it in action, see the source file `three_way_if.f90`. Both statement types still exist in Fortran, or at least in the compilers, to support old-style programs.[3]

3.4 Jumping to the end

The `GOTO` statement was and is also used to jump to a completely different part of the program unit for reporting error conditions:

```
      SUBROUTINE PRSQRT( X )

      IF ( X .LT. 0.0 ) GOTO 900

      WRITE(*,*) 'Square root of X = ', X, ' is ', SQRT(X)
      RETURN

900   CONTINUE
      WRITE(*,*) 'X should not be negative - ', X
      STOP
```

[3]The arithmetic IF statement was deleted from the Fortran 2018 standard, as gfortran will report. Intel Fortran will tell you this when you specify the standard as f18 – "-stand", defaulting to the Fortran 2018 standard.

```
      END
```

Such statements can be gathered at the end of the program unit so as not to clotter the code that deal with normal processing. If you want to avoid the `GOTO` statement, then the modern `BLOCK` construct will help:

```
subroutine print_sqrt( x )
    real :: x


    normal: block
        if ( x < 0.0 ) exit normal

        write(*,*) 'square root of x = ', x, ' is ', sqrt(x)
        return
    end block normal
    !
    ! Error processing
    !
    errors: block
        write(*,*) 'x should not be negative - ', x
        stop
    end block errors

end subroutine print_sqrt
```

Note to self: http://www.u.arizona.edu/ñubinson/copyright_violations/Go_To_Considered_Harmful.html Reprinted from Communications of the ACM, Vol. 11, No. 3, March 1968, pp. 147-148. Copyright (c) 1968, Association for Computing Machinery, Inc.

3.5 The ASSIGN statement

The uses of `GOTO` so far have all been static: the statement labels were fixed in the code. While the `GOTO` statement is frowned upon and it can certainly make the control flow difficult to follow when you do not use it in an orderly fashion, there is a possibility to use "dynamic" labels, so that the `GOTO` effectively jumps to a varying location. This is achieved by the `ASSIGN` statement. It is not often used, as in most cases better and especially clearer constructs are possible, even in FORTRAN, but here is one possible case:
Suppose you have to compute something complicated in a number of places in a program unit, based on a large number of variables, so that using a subroutine or a function is awkward, as it leads to a very long argument list. Nowadays you can easily use an internal routine, but this was not the case with FORTRAN. So, with the `ASSIGN` statement you could store a location to rerturn to, jump ahead to the complicated piece of code, jump back when done, and remain in the same routine. Here is a simple example:

```
      SUBROUTINE( ICASE )
      A = 1.0
      B = 2.0
      C = 3.0
      IF ( ICASE .EQ. 1 ) ASSIGN 100 TO JMP
      IF ( ICASE .EQ. 2 ) ASSIGN 200 TO JMP
      IF ( ICASE .EQ. 3 ) ASSIGN 300 TO JMP
      GOTO 900

* Case 1: use the result in F
  100 CONTINUE
      WRITE(*,*) 'Case ', ICASE, 'value is ', F
      GOTO 400

* Case 2: use the result in G
  200 CONTINUE
      C = 4.0
      WRITE(*,*) 'Case ', ICASE, 'value is ', G
      GOTO 400

* Case 3: use the result in H
  300 CONTINUE
      WRITE(*,*) 'Case ', ICASE, 'value is ', H
      GOTO 400

* All done, continue
  400 CONTINUE
      WRITE(*,*) 'Done'
      RETURN

*
  900 CONTINUE
* We can use the local variables directly
      F = A + B + C
      G = A + B * C
      H = A * B + C

* We are done, so return to the "caller"
      GOTO JMP
      END
```

It is a useless and contrived example, but it is only meant to illustrate how a "dynamic" jump can be constructed – the part after statement label 900 is actually independent of whatever happens above it. You can extend it with new cases, without having to worry about the computational part.

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.[4]

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 scarcer 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.

  In the code that would look like:

  ```
  COMMON /ABC/ WORK(1000)
  EQUIVALENCE (WORK(1), IWORK(1))
  ```

  with a typical sloppiness with respect to array dimensions.

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 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 particular use was an extension to the standard.

4.2 More on EQUIVALENCE

TODO

4.3 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:

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

```
      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.
>
> Some compilers have an option to enforce the `SAVE` attributes on variables, irrespective of the source code. You should take special care if old source code relies on such an option.

4.4 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 and thus the executable itself. The following program leads to an executable of approximately 5.7 MB using gfortran on Windows:

```
program data_stmt
    implicit none

    integer :: i
    real    :: array(1000000)

    data (array(i), i = 1,size(array),2) / 500000*1.0 /
   data (array(i), i = 2,size(array),2) / 500000*2.0 /

    !
    ! Alternative
    !
    !! array(1::2) = 1.0
    !! array(2::2) = 2.0
    !

    write(*,*) sum(array)
end program data_stmt
```

If, instead, you use the alternative and remove the `DATA` statements, the executable is only 1.7 MB. Of course, this is an exaggerated example, but it illustrates that such `DATA` statements are very different in character than ordinary, executable statements.

4.5 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 usually 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 and is not part of a routine or the main program. 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:[5]

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

Both build commands succeed, despite the fact that one part of the program is missing in the second one.

The `block.f90` source file is:

```
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.

4.6 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 )
*
```

[5] 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.

```
*  -- 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:[6]

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

[6] Intentionally left in fixed form.

```
*     ..
```

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

> The careful reader wil note that one feature of the original interface has not been retained in the simplification: `NHRS`. Thus, with this revised interface the array `B` should consist entirely of right-hand side vectors.

5 Subjects

```
- array(*) versus array(:)
- array(10) as the starting point
- 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
- 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
- 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 (?)
- use of d00 in input
- Cray pointers
- uppercase/lowercase letters
- D as comment character
- external, also notation "*tan"
- FORTRAN 66 semantics: OPEN - STATUS = 'NEW' as default.
```

- Effect of BLANK = 'ZERO' versus BLANK = 'NULL'
- Specific names for functions like max() and sin()
- Alternate return

References to be added: IEEE 754 and Fortran 90 standard.