Old programming idioms explained

Arjen Markus

July 14, 2023

Contents

1 Introduction **3**

2 Source forms **3**
2.1 Fixed form versus free form 3
2.2 The significance of spaces 4
2.3 The character set 5
2.4 Variables, functions, names and types 5
2.5 Special comments: D 6
2.6 Compiler directives 7
2.7 Separate compilation 8

3 Subroutines and functions **8**
3.1 Passing arrays 9
3.2 Specific names for intrinsic functions 10
3.2.1 A curious function: DIM 11
3.3 The EXTERNAL statement 11
3.4 Statement functions 12
3.5 Constant values as actual arguments 13
3.6 The ENTRY statement 13
3.7 Alternative returns 15

4 Floating-point numbers **15**
4.1 REAL*4 and REAL*8 16
4.2 Literals in the source code 17
4.3 Input in the absence of a decimal point 18

5 Handling characters **18**
5.1 Hollerith constants 18
5.2 Fixed-length character arguments 20
5.3 Interfacing to C: hidden arguments 21

6 Control structures **22**
6.1 Ordinary and nested DO-loops 22
6.2 Simulating a DO-WHILE loop 25
6.3 Three-way IF statements and computed GOTOs 27
6.4 Jumping to the end 28
6.5 The ASSIGN statement 29
6.6 The DO-loop with a real index variable 30

7 Memory management **31**
7.1 COMMON blocks 32
7.1.1 More on EQUIVALENCE 33
7.2 The SAVE statement 34
7.3 The initial values of (local) variables 34
7.4 Initialising variables in COMMON blocks: BLOCK DATA 36
7.5 Work arrays 38

8 Input and output intricacies **39**
8.1 Standard input and output 40
8.1.1 Carriage control 41
8.2 Direct-access files 41
8.3 Unformatted sequential files 42
8.4 Reading an array 45

9 Some caveats for programmers coming from C-like languages **45**
9.1 Terminology and semantics 45
9.2 Subroutines and functions 46
9.3 Initialisation of variables 46
9.4 Handling character strings 47

10 Subjects **48**

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.

Of course, a document like this can hardly be complete but hopefully it is useful enough.

2 Source forms

The source form of FORTRAN, now known as *fixed form*, shows its heritage in the era of punchcards: Each line could be up to 80 characters long, but only the first 72 characters had actual meaning. Some editors would add line numbers in the column 73–80 or people could add short comments. It had and has a few curiosities beyond the mere signficance of the columns.

The alternative source form that was introduced with the Fortran 90 standard is more flexible and should definitely be used for any new source.

2.1 Fixed form versus free form

FORTRAN source files usually have an extension `.f` and on PCs, before you could use long names, the extension often was `.for`. There is, however, nothing special about these file extensions. They are merely a convention, useful for the compiler as it can use it to identify the source form. The free form source is often indicated by the extension `.f90`. *This should not be taken to mean that the code adheres to the Fortran 90 standard*, just as there is no particular difference as far as file extensions are concerned for FORTRAN 66 or FORTRAN 77.[1]

To elaborate a bit on these file extensions:

- Fixed form is often identified by `.f` or for some compilers on Windows `.for`, but that is not mandated by any standard and compilers often support options to specify what the source form is, if the extension is misleading.

[1] File extensions themselves are a fairly recent feature of computer file systems. Older systems had different conventions.

- On file systems where file names are case-sensitive, such as Linux, an extension `.F` or `.F90` is often meant to automatically invoke a preprocessor like C's preprocessor. Alternatively, you can also use compiler options to invoke the preprocessor explicitly.
- Preprocessing is not part of the FORTRAN or Fortran standards. It is simply an extension (no pun intended) that may or may not be supported by the compiler. As a consequence, there is no prescribed syntax, though very often the C conventions are used.

2.2 The significance of spaces

The original fixed form for FORTRAN sources has at least the following curiosities:

- In the columns 1 to 5 you can only put comments, if the first character is a comment character ("C" or "*"), or statement labels.
- The sixth column is reserved to indicate that the previous line is continued:

```
*234567890
      WRITE(*,*) 'A long sentence that spans over two',
     &           'or more lines'
```

  The continuation character, in the above the "&", can actually be any character with the exception of a zero (0). Some programmers would use digits, so that you can easily count the number of lines.
- The columns 7 to 72 were used for the actual code.
- The columns 73 to 80 were reserved for comments, to the discretion of the user.
- *Spaces had no significance*, except within literal strings.

Especially this last property may come as a surprise. To illustrate this (see `fixedform.f`):

```
      PROGRAM FIXEDFORM

      INTEGER :: I

      DO 100 I = 1.10
          WRITE(*,*) I
  100 CONTINUE
!
!     ILLUSTRATE THE FACT THAT SPACES HAVE NO MEANING ...
!
      DO 110 I = 1,10
```

```
      I F ( I . E Q . 4 ) T H E N
          W R I T E ( * , * ) I
      E N D I F
  110 C O N T I N U E
      END
```

Keywords in FORTRAN and Fortran are not reserved words and in fixed form you can put between the letters as many spaces you want, anywhere.

Running this program might produce the following output:

```
578772136
       4
```

"Might", because there is an uninitialised variable here: the line `DO 100 I = 1.100` contains a typo. Instead of a comma, it contains a decimal point and thus the line is interpreted as:

```
DO100I = 1.10
```

The variable `DO100I` gets assigned the value 1.10, which is why there is no DO loop to produce ten lines in the output, counting from 1 to 10. Such mistakes can be caught by using the `IMPLICIT NONE` statement, a common enough extension.[**?**][2]

Note to self: https://stevelionel.com/drfortran/2021/09/18/doctor-fortran-in-implicit-dissent/

2.3 The character set

Officially, FORTRAN code should be written with capitals only. Lowercase letters are only allowed in literal strings. Again, a heritage of the machinery of old. But it has been allowed by compilers to use lowercase as well.

Beyond letters of the Latin alphabet you can use digits and underscores and several other characters. Even today, the standard is quite strict: characters outside the officially supported set are only allowed (or tolerated?) in literal strings.

2.4 Variables, functions, names and types

A feature that Fortran code relies on less and less is the use of *implicit typing*: of old a variable, or indeed a function, was of type `integer` if its name starts with one of: I, J, K, L, M or N. If not, the variable or function was of type `real`. You could declare the name as being of a different type, but that required an explicit declaration.

Because of this implicit typing, mistakes in names were a serious source of errors.

In FORTRAN, names were also limited to six (significant) characters, a limitation shared with the linkers.

[2] This statement was formalized in "MIL-STD-1753", precisely to make the coding safer.

This six-characters limit was even more severe as the names of subroutines, functions and `COMMON` blocks were global for the whole program. There was no other scope or control of visibility. This made it very important to explicitly define the names of such entities for any library you wrote, as naming conflicts could not be solved via a renaming clause, as they can nowadays in Fortran.

While the `IMPLICIT` statement is used nowadays mostly to turn off the implicit typing via `implicit none`, it can be used to define a particular type to variable whose names started with a particular character:

```
IMPLICIT DOUBLE PRECISION (A-H,O-Z)
```

means that all variables (and functions) whose names start with any of the letters A, B, ..., H or O, P, ... Z would have the type `DOUBLE PRECISION`, unless explicitly given a different type.

2.5 Special comments: D

Sometimes in old code you can see a character "D" in the first column. This is considered a comment line, unless you specify a compile option like `-d-lines` for the Intel Fortran compilers. Not all compilers support this, gfortran does not seem to have an option for it. The effect of specifying the relevant option is that the line is no longer considered a comment line, but is actual code. If supported by the compiler, it is only available for fixed form source.

It is a rather awkward form of "preprocessing":

- It is a compiler extension or at least not supported by all compilers. The now deprecated g77 compiler acknowledged the existence of such lines, but provided no facilities.

  Note to self: https://gcc.gnu.org/onlinedocs/gcc-3.4.6/g77/Debug-Line.html

- It is not available for free form source.

- A practical problem is that it does not evolve with the code itself, as in the normal build process, these lines are considered comments.

It is just as easy to use ordinary code to provide debugging facilities:

```
*234567890
      PROGRAM FIXEDDEBUG

D     WRITE(*,*) 'Note: in debugging mode ...'
      WRITE(*,*) 'Hello, world'

      END PROGRAM FIXEDDEBUG
```

The program (see the file `fixeddebug.f`) is not accepted by gfortran because of the D-line, but Intel Fortran compiles it with and without the option `-d-lines`, only producing different output.

You could make the intent clearer using something along these lines (or a modern equivalent):

```
PROGRAM FIXEDDEBUG
LOGICAL DEBUG
PARAMETER (DEBUG = .FALSE.)

IF (DEBUG) WRITE(*,*) 'Note: in debugging mode ...'
WRITE(*,*) 'Hello, world'

END PROGRAM FIXEDDEBUG
```

With FORTRAN the `debug` parameter was best put in an include file to ensure that every program unit used the same setting. With Fortran a module containing such a parameter will do.

> *Note:* If you use a *parameter* instead of a *variable*, many compilers will even eliminate the debugging code from the resulting program, but the code is compiled.

2.6 Compiler directives

Besides specific options, most if not all compilers have what are known as *compiler directives*. These take the form of a comment with a special format. As they are specific to the compiler, the precise form depends on the compiler, as well as what you can do with them. There are also compiler directives that are defined as a standard, like the *OpenMP* directives.[3]

Here is one example: an implicit way to redefine the default precision for real variables:

```
C For the Intel Fortran compiler:
!DIR$ REAL:8
```

This specifies the `kind` of real variables that are not declared with an explicit kind themselves.

Another example of a completely different nature:

```
C For the gfortran compiler:
!GCC$ unroll 10
```

The gfortran compiler supports loop unrolling and with this directive you control how this is done.

Note to self:

https://gcc.gnu.org/onlinedocs/gfortran/UNROLL-directive.html

https://www.intel.com/content/www/us/en/docs/fortran-compiler/developer-guide-reference/2023-0/real-directive.html#GUID-38B61FE4-4058-497A-9AAA-2AF1BB56C178

[3] Compiler directives have their equivalent in C. There they are called pragmas and work in much the same way.

Since these directives come with a specific compiler, you will need to check the documentation of the program or, more generally perhaps, the favoured compiler to see what the meaning is. It is not always entirely innocent: loop unrolling is an optimisation technique, but barring errors in the compiler, it should not matter for the outcome of a program whether the directive has been adhered to or not. This is different for the first directive – the default precision may have a significant effect!

2.7 Separate compilation

One of the objectives of the design of FORTRAN, which continues in today's standards, is that source files can be compiled independently. This shoud be read as: it is not necessary to compile the whole program in one step. In FORTRAN that was fairly easy: besides include statements there is no actual dependency possible. So, a source file can be compiled without knowing anything about other parts of the program.

In Fortran, there are dependencies beyond included files: if the program unit uses one or more (non-intrinsic) modules, then the order of compilation is that first the source files containing these modules must be compiled and then the source file with the using program unit.

The idea of separate compilation is a very powerful one, certainly in times when desk checking the code was a necessary first step, because the actual compilation might be done during the night, with you coming back in the morning to find out that you made a typo somewhere. Compiling source files separately and storing the resulting object files in libraries helps to minimize the risks of losing time.

But a side effect of this is that some programmers interpreted this as meaning: you should put all subroutines and functions in separate files. This does have one advantage, namely that the object files would only contain a single program unit and the linker would incorporate only the program units that are actually called. The disadvantage is that you may have to combine a large number of files into a library, because all the little subroutines and functions are in individual source files.

With the modules we have nowadays it is much easier to manage the routines. You can hide the ones that are not intended for outside use (so their names cannot cause trouble) and you can store the routines that functionally belong together in one and the same module, or organise the code in submodules.

3 Subroutines and functions

Older compilers were not so very sophisticated when it comes to checking the actual argument lists of calls to subroutines or functions against the dummy (expected) argument lists of these program units. This is partly due to there not being a mechanism in FORTRAN for providing such checks, even though some compilers could implement these checks at run-time. It was also used

sometimes to allow for flexibility. For instance: if you do not care if an array holds integers or reals, because you are simply writing them to some binary file, then one routine would suffice to take care of both types (perhaps even including double precision numbers).

NOTE: example

A more common situation: allow one-dimensional arrays to be used as two-dimensional or pass scalars to array dummy arguments. As long as you remain with the allotted memory, there should be no problem.

NOTE: example

3.1 Passing arrays

Several surprising idioms are connected to the passing of arrays to subroutines and functions:

- The dummy argument could be declared as `REAL ARRAY(1)`. This was not unusual in programs predating FORTRAN 77, as that introduced the asterisk for arrays whose sizes are not defined in the routine perse: `REAL ARRAY(*)`, the so-called assumed-size arrays. A small example:

```
      SUBROUTINE PRINT( A, N )
      INTEGER N
C
C     Array declared pre FORTRAN 77:
C
      REAL A(1)
C
C     Array declared the FORTRAN 77 way:
C
C     REAL A(*)
      ...
      END
```

  In both cases, the routine would have to know what the actual size of the array should be. Often, that was done via an extra argument like the argument `N` in the example (see for a more elaborate and practical example section 7.5).

  With Fortran you can use assumed-shape arrays, so that the dimensions of the array are passed on automatically. This does require that the interface is explicit. If the compiler does not see that interface, it has to assume that the FORTRAN 77 style is to be applied.

- The actual argument could be some element of the array:

```
      REAL ARRAY(100)
```

```
CALL PRINT( ARRAY(10), 30 )
END

SUBROUTINE PRINT( A, N )
INTEGER N
REAL A(N)
WRITE(*,*) A
END
```

The call to `PRINT` passes the array elements 10, 11, ... of `ARRAY` to the subroutine and in the subroutine it is assumed that the array `A` (its first dummy argument) is `N` (it second dummy argument) elements large. Then we can print the dummy array as a whole.

With Fortran we can specify a section of an array, not necessarily a contiguous section:

```
real :: array(100)

call print( array(10::10) )
```

The array section selects elements 10, 20, ... 100, but as the subroutine `PRINT` is in old-style FORTRAN, it does not get an *array descriptor*. The compiler will have to make a temporary array, copy in the selected elements, call `PRINT` and before releasing that temporary array, copy the elements of that temporary array into the selected elements. *It does not know the intent of the arguments of the subroutine* `PRINT` *after all!*

Marking the *intent* of arguments has at least two advantages: documenting the intended use of the argument and making certain optimisations possible. Like in the above example: copying back is not necessary if the argument is `intent(in)`.

3.2 Specific names for intrinsic functions

With the advent of FORTRAN 77 the language gained generic names of such intrinsic functions as `MAX` and `SIN`. Pre-FORTRAN 77 code uses specific names to distinguish the types of the arguments and the result. Thus:

- The minimum and maximum functions for integers were `MIN0` and `MAX0`.
- The minimum and maximum functions for single-precision reals were `AMIN1` and `AMAX1`.
- The minimum and maximum functions for double-precision reals were `DMIN1` and `DMAX1`.
- If you wanted an *integer* result from a list of *reals*, then the function names were `MIN1` and `MAX1`. But a *real* result from a list of *integers* is returned by `AMIN0` and `AMAX0`.

- For the trigonometric functions: `SIN`, `DSIN` and `CSIN` would return the sine of respectively a single-precision real, a double-precision real and a complex value.

The Internet, as usual, will provide detailed information on the various intrinsic functions.

Sometimes, these specific names are required if you want to pass them as arguments to a routine. The best way is to avoid doing so and write a small helper function instead, as these specific names are deprecated and may even be removed from the standard altogether.

3.2.1 A curious function: DIM

Here is a function that is defined by the FORTRAN 77 standard but whose usefulness seems questionable – at least I have never actually seen it used:

```
C     If X > Y, then return the difference, otherwise zero.
      POSDIF = DIM(X,Y)
```

This is merely to warn that other functions and subroutines may be found in old FORTRAN code that have lost their meaning or have been replaced by others. Compiler-specific procedures that have been replaced by standard facilities include `SYSTEM` to run an external program and various functions to get the system time.

3.3 The EXTERNAL statement

Consider the following program:

```
      PROGRAM USEFNC
      REAL A
      n = 10
      WRITE(*,*) A(N)

      END
```

You cannot build it by itself, because it refers to a *function* `A` which is external to the program. You get an error about an unresolved symbol or a similar message. If you add the definition of a function `A` to the build step, then, of course, that symbol is resolved and the program can be built.

But the more important thing to notice is that the call to `A` is indistinguishable from using `A` as an array. To make the distinction clear, you can use the `EXTERNAL` statement. This is also useful for arguments you pass that are actually procedures (see `integral.f`):

```
      PROGRAM CALC
      REAL A, INTGRL
      EXTERNAL A, INTGRL
```

```
      XBEGIN = 0.0
      XEND   = 1.0
      WRITE(*,*) INTGRL(A, XBEGIN, XEND)
      END

      FUNCTION A(X)
      REAL X
      A = X ** 2
      END

      REAL FUNCTION INTGRL(A, XB, XE)
      REAL A, XB, XE
      EXTERNAL A
      REAL DX

      DX     = (XE - XB) / 10.0
      INTGRL = 0.0
      DO 100 I = 1,10
          X      = XBEGIN + (I-1) * DX
          INTGRL = INTGRL + A(X)
  100 CONTINUE

      INTGRL = INTGRL * DX
      END
```

Note that the declaration of `INTGRL` as a *real* function is required because the implicit typing rules would otherwise make it an integer, with very strange results.

NOTE: EXTERNAL for an intrinsic procedure

3.4 Statement functions

Short functions like the function `A` in the previous section can also be written as so-called *statement functions*. These appear among the general declarations:

```
      PROGRAM CALC
      REAL INTGRL
C
C Use a statement function instead of a regular function
C
      A(X) = X ** 2

      XBEGIN = 0.0
      XEND   = 1.0

      INTGRL = 0.0
```

```
      DO 100 I = 1,10
          X      = XBEGIN + (I-1) * DX
          INTGRL = INTGRL + A(X)
  100 CONTINUE

      INTGRL = INTGRL * DX
      WRITE(*,*) INTGRL(A, XBEGIN, XEND)
      END
```

As a statement function is not allowed as an actual argument, the integration is now moved into the program itself instead of being implemented in a separate function.

Statement functions can be useful to abbreviate expressions that appear at multiple locations in the code of a routine, but *internal procedures* in Fortran are much more useful and recognisable.

3.5 Constant values as actual arguments

A peculiar phenomenon was possible in the old days: you could change the value of a literal constant. That is: if you were sloppy, you could change the value of, say, `10` into `20`:

```
      PROGRAM CONST

      CALL DOUBLE( 10 )
      WRITE(*,*) 10
      END

      SUBROUTINE DOUBLE( X )
      INTEGER X

      X = 2 * X
      END
```

Nowadays, compilers will not pass the literal constant itself but a copy or something similar, so that the above example will do no harm. But if the implementation is such that `10` is actually a memory address that is readable and writeable, then havoc is raised with, as a bonus, difficult to track bugs.

3.6 The ENTRY statement

FORTRAN offered the `ENTRY` statement to provide alternative entries into a subroutine or function. Such statement makes it possible to call the procedure with a different argument list, for instance. A slightly contrived example:

```
      REAL FUNCTION SETACC( X, START )
```

```
      REAL X, START

      REAL SUM
      SAVE SUM

      SUM    = START + X
      SETACC = SUM
      RETURN

      ENTRY NXTACC( X )
      REAL X

C
C     Accumulate to the variable SUM defined in SETACC
C
      SUM    = SUM + X
      NXTACC = SUM
      END
```

With the funcion `SETACC` you initialise an "accumulator" variable and with `NXTACC` you add a new value to it and get the result. In a program you could use it like this (see `accum.f`):

```
      PROGRAM ACCUM
      REAL X, SUM

      OPEN( 10, FILE = 'accum.inp' )

      READ( 10, * ) X
      SUM = SETACC( X, 0.0 )

  100 CONTINUE
      READ( 10, *, END = 110, ERR = 110 ) X
      SUM = NXTACC( X )
      GOTO 100

  110 CONTINUE
      WRITE(*,*) 'Sum: ', SUM
      END
```

There are doubtless other uses for `ENTRY`, but the Fortran standard offers modules and module variables for this sort of things instead.

As the `ENTRY` statement only defines the name and the argument list, it serves as a subroutine, if its corresponding program unit is a subroutine or as a function of the same type. Its argument list can, however, differ.

3.7 Alternative returns

A deprecated feature in Fortran, alternative returns provide a means to transfer the program control from a routine to its caller to another statement than the one following. This feature could be used for error handling (see `errors.f`):

```
      PROGRAM ERRORS
      REAL X

      X = 2.0
      CALL PRSQRT( X, *900 )

      X = -1.0
      CALL PRSQRT( X, *900 )
      STOP
C
C     Handle any errors
C
  900 CONTINUE
      WRITE(*,*) 'Routine PRSQRT expects a positive argument'

      END

      SUBROUTINE PRSQRT( X, * )
      REAL X

      IF ( X .GE. 0.0 ) THEN
          WRITE(*,*) 'The square root of ', X, ' is ', SQRT(X)
      ELSE
          RETURN 1
      ENDIF
      END
```

Note: because the feature is deprecated (obsolescent), compilers may not properly check that the actual call has the right number of statement labels in the interface. In most cases a simple return code works quite as well (the section 6.4 offers more on the subject).

4 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.

4.1 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.[4]

While FORTRAN has supported complex numbers for a very long time, it did not standardise double-precision complex numbers. Compiler extensions like `COMPLEX*16` often filled that gap. With Fortran, the *kind* feature is used to achieve this:

```
integer, parameter :: dp = kind(1.d0)
complex(kind=dp)   :: z
```

The kind refers to the underlying floating-point number, not the storage, like with the `COMPLEX*16` type.

> *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

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

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.

4.2 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):

```
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.[5]

[5] In general, using these compile options is considered a bad idea. It is all too easy to forget them when building the program or library.

4.3 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 the 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.

5 Handling characters

With the FORTRAN 77 character strings were introduced. Before that there was very little that could be done with characters, unless they represented numbers. It was the era of the hollerith constants. But FORTRAN 77 only introduced fixed-length strings. Only with the advent of Fortran 2003 did the language acquire the possibility to work with strings of dynamically allocatable lengths.

5.1 Hollerith constants

The main use you are likely to encounter of hollerith constants is in format statements or as constant strings passed to a subroutine. Here is a code fragment from a library for special functions, like the Bessel functions and many others:

```
c
      if (ntk0.ne.0) go to 10
      ntk0 = inits (bk0cs, 11, 0.1*r1mach(3))
      xsml = sqrt (4.0*r1mach(3))
      xmax = -log(r1mach(1))
      xmax = xmax - 0.5*xmax*log(xmax)/(xmax+0.5) - 0.01
c
 10   if (x.le.0.) call seteru (29hbesk0   x is zero or negative, 29,
     1  2, 2)
      if (x.gt.2.) go to 20
```

The construction `29hbesk0   x is zero or negative` is a hollerith constant, in this case, it is 29 characters long and passed to a subroutine that will print an error message. The signature of the subroutine is:

```
      subroutine seteru (messg, nmessg, nerr, iopt)
      common /cseter/ iunflo
      integer messg(1)
      data iunflo / 0 /
```

So the hollerith string is stored in an integer array `messg`, but note:

- The size of the array is defined as 1, because there was no concept of an assumed-size array before FORTRAN 77 (see section 3.1).
- The variable `iunfo` is apparently initialised in an ordinary `DATA` statement, even though it is contained in a `COMMON` block (see section 7.4).
- The code is written in lower-case. Strictly speaking that is not conforming to the standard.
- The array `messg` in turn is passed onto another routine, `seterr`, but the code for that routine simple prints "Error!" and we are not shown how to deal with the integer array that holds the character string.

Modernising holleriths that only serve as static strings is simple: turn them into ordinary strings. However, sometimes they were used to manipulate string data and then you need to understand exactly what is going on, especially because vendor-specific methods were typically used and the storage of hollerith constants was non-portable [**?**].

Note to self: https://en.wikipedia.org/wiki/Hollerith_constant

An example of manipulating character strings pre-FORTRAN 77 is given here, shamelessly copied from a Fortran discourse thread. The code fragment uses compiler-specific statements `DECODE` and `ENCODE` to move characters to and from integers:

```
      CHARACTER S*6 / '987654' /, T*6
      INTEGER V(3)*4
      DECODE( 6, '(3I2)', S ) V
```

```
      WRITE( *, '(3I3)') V
      ENCODE( 6, '(3I2)', T ) V(3), V(2), V(1)
      PRINT *, T
      END
```

As stated in the thread:

> The above program has this output:
>
> ```
> 98 76 54
> 547698
> ```
>
> The DECODE reads the characters of S as 3 integers, and stores them into V(1), V(2), and V(3). The ENCODE statement writes the values V(3), V(2), and V(1) into T as characters; T then contains '547698'.

Note to self: https://fortran-lang.discourse.group/t/code-that-baffles-me-could-someone-please-explain/6005

5.2 Fixed-length character arguments

Just as with arrays you specify a fixed length for character arguments. The argument will act as if it actually holds strings of the specified length, but other than with arrays, a mismatch with the actual argument may lead to strange results. Here is a small example:

```
C234567
      PROGRAM SHWSTR
      CHARACTER*10 STRING(5)

      STRING(1) = 'ABCDEFGHIJ'
      STRING(2) = '1234567890'
      STRING(3) = '?         '
      STRING(4) = '?         '

      CALL STR3(  STRING, 2 )
      CALL STR13( STRING, 2 )
      END
      SUBROUTINE STR3( STRING, N )
      CHARACTER*3 STRING(N)
      INTEGER I
      DO I = 1,N
          WRITE(*,*) I, STRING(I)
      ENDDO
      END
      SUBROUTINE STR13( STRING, N )
```

```
      CHARACTER*13 STRING(N)
      INTEGER I
      DO I = 1,N
          WRITE(*,*) I, STRING(I)
      ENDDO
      END
```

The output of the program is:

```
 1 ABC
 2 DEF
 1 ABCDEFGHIJ123
 2 4567890?
```

Since character strings are laid out in memory in a contiguous fashion, the mismatch causes the strings as seen in the subroutines to be just strings of the given length, *independent therefore* of the strings in the calling program unit.

In general, it is best to use assumed-length strings: `CHARACTER*(*) STRING` or, as Fortran would have it: `character(len=*) :: string`.

5.3 Interfacing to C: hidden arguments

Since a subroutine or function that has one or more character arguments must somehow "know" the length of the actual arguments (otherwise the assumed-length declaration would not work), that information is passed from the caller to the callee. The method is not prescribed by the standard, but there exist at least two methods, which are both invisible from the FORTRAN side. But if you interface with C routines, then it becomes important to know which one is actually used: on the C side a hidden argument that gives the length is visible and this extra argument is either directly after the character argument or after all the regular arguments.

This is one of the difficulties that existed in FORTRAN in dealing with other languages, notably C.

Here is an illustration. The program calls a C routine that accepts a character string array, much as in the previous section:

```
C234567
      PROGRAM SHWSTR
      CHARACTER*10 STRING(5)

      STRING(1) = 'ABCDEFGHIJ'
      STRING(2) = '1234567890'
      STRING(3) = '?         '
      STRING(4) = '?         '

      CALL STRC(  STRING, 2 )
      END
```

The C routine does very little, but looks like this (the declaration is more important than the body), if the hidden argument comes right after the string argument:

```
/* Hidden argument directly after the associated argument */
#define STRC ...
void STRC( char *string, int lenstr, int *n ) {
    ...
}
```

and otherwise like this:

```
/* Hidden argument after all ordinary arguments */
#define STRC ...
void STRC( char *string, int *n, int lenstr ) {
    ...
}
```

Note that:

- We define the name via a macro, `STRC`, because the actual name depends on the FORTRAN compiler. With Fortran 2003, this problem is solved via the `bind(C)` attribute, with or without the `name = "..."` clause.
- The hidden argument is a *value*, not a pointer like the size parameter n. In FORTRAN such arguments cannot be expressed, so that more often than not the interfacing with C requires an intermediate "wrapper" routine to translate such incompatibilities. Again, Fortran 2003 solves much of this in a standardized way.

6 Control structures

FORTRAN 77 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 arithmetic (or three-way) `IF` and the computed `GOTO`, as well as the `ASSIGN` statement. This part of the document highlights these ancient idioms.

6.1 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
          SUM = SUM + ARRAY(I)
  110 CONTINUE
  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: there were a large number of constraints and as a curious interaction with the compilers not detecting that these constraints were violated, quite often a `DO` loop would run at least once. Consider this code:

```
* With the right compiler options, print this line once!
* (Note: FORTRAN 77, not strictly FORTRAN 66)
      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 this "feature",

As Ron Shepard explains on Fortran discourse:

> F66 do loops were not directly or specifically required to execute once, rather the range parameters were restricted so that a single pass was always executed for conforming code. The one-trip execution feature followed indirectly from these constraints. The loop in the example code
>
> ```
> do i = 1,0
>     write(*,*) 'FORTRAN 66: ', i
> enddo
>
> write(*,*) 'Current value of i:', i
> ```
>
> would have been illegal for five reasons. 1) the do statement would have required an integer statement label; the unlabeled do with enddo was not introduced until f90. 2) the termination value m2 is zero. In f66, the m1, m2, and m3 values were all required to be

greater than zero. 3) the m1 value must be less than or equal to the m2 value, a constraint on the programmer that is violated in this example. 4) the enddo statement did not exist in f66, it would have been a labeled continue statement. 5) upon execution of the loop, the loop control variable i is undefined, so the final write statement is referencing an undefined integer.

Many compilers would not diagnose and detect violations to the m1, m2, m3 constraints, so they would execute the loop with a single pass. But that was not required or specified by the standard, the standard simply stated the requirements which were violated by the programmers in these cases.

Of course, the write statements also violate f66 in other ways. 1) Character constants did not exist until f77. 2) List-directed i/o did not exist until f77. 3) The use of * to specify the default output logical unit was not defined until f77.

Some compilers still provide an option to allow for the FORTRAN 66 semantics,[**?**] which includes this feature:[6]

6.2 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
  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
```

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

```
      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

6.3 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.[7]

6.4 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
      END
```

Such statements can be gathered at the end of the program unit so as not to clotter the code that deals 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
```

[7] 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.

```
        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: the `errors` block is merely introduced to syntactically distinguish the error handling from the normal handling. It does not influence in this form the flow of control.

Note to self: http://www.u.arizona.edu/~rubinson/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.

6.5 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 return 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 )
      SAVE
      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.

6.6 The DO-loop with a real index variable

A peculiar feature of FORTRAN 77 was the introduction of a DO-loop governed by a real variable. This means that the precise values of the variable might influnce the number of iterations, for instance. A simple program to illustrate this (see `real_do.f90`):

```
      PROGRAM REAL_DO
```

```
IMPLICIT NONE

INTEGER I, N
REAL    R

DO 120 I = 1,10
    N = 0
    DO 110 R = 0.0, 1.0*I, 0.1*I
        WRITE(*,*) R
        N = N + 1
    ENDDO
    WRITE(*,*) 'Number of iterations: ', N
ENDDO
END
```

The output of the program with the Intel Fortran (leaving out the values of R):

```
 Number of iterations:           11
 Number of iterations:           11
 Number of iterations:           10
 Number of iterations:           11
 Number of iterations:           11
 Number of iterations:           10
 Number of iterations:           11
 Number of iterations:           11
 Number of iterations:           10
 Number of iterations:           11
```

So, some DO-loops get 11 iterations, what you would expect with exact arithmetic and others get only 10. If you run this program with the gfortran compiler the result is a uniform 11 iterations – apparently the loop is controlled in a different way! Apart from the unpredictability of the number of iterations with one compiler, you can also not rely on the portability of the program.

7 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.

7.1 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.[8]

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.

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

- 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.

7.1.1 More on EQUIVALENCE

A specific use of the `EQUIVALENCE` statement is to access the binary representation of a (floating-point) number. A concrete example is the conversion of big-endian numbers to little-endian numbers or vice versa:

```
      INTEGER      A, B
      CHARACTER*4  BIG, LITTLE
      EQUIVALENCE (A, BIG)
      EQUIVALENCE (B, LITTLE)
C
C     Read some number from a file - big endian
C     and rearrange the bytes
C
      READ(10) A
      LITTLE(1:1) = BIG(4:4)
      LITTLE(2:2) = BIG(3:3)
      LITTLE(3:3) = BIG(2:2)
      LITTLE(4:4) = BIG(1:1)

      WRITE(*,*) 'Value is: ', B
```

7.2 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.
>
> 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.

7.3 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 a somewhat similar 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.

7.4 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 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:[9]

```
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.

[9] 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.

7.5 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:[10]

[10] Intentionally left in fixed form.

```
      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.

> 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.

8 Input and output intricacies

Placeholder:

```
- command-line arguments for file names
- big-endian and little-endian
- list-directed input and output - also: /
- narrow formats (?)
- use of d00 in input
- FORTRAN 66 semantics: OPEN - STATUS = 'NEW' as default.
- Effect of BLANK = 'ZERO' versus BLANK = 'NULL'
- 32-bits machines and unformatted files
```

Almost any program will read some kind of input files and produce some kind of output files. FORTRAN defined, roughly, three types of files:

- Text files meant to read or edited by humans. These are known as *formatted files*.
- Binary files with a record structure of sorts, so that you could read a part of the record and then automatically jump to the next. These are *unformatted sequential files*. They are compact and might be considered *binary* files.
- Binary files with records that have a fixed length and where you can position the `READ` or `WRITE` action to a particular record: the *unformatted direct-access files*.[11]

[11] Actually, there are also formatted direct-access files, but these are seldom used.

Input and output in FORTRAN was always oriented towards records. For instance, if you read a number from a formatted file, then the read position is automatically moved to the start of the next line, independent of the amount of data left on the previous line.

Similarly, writing to a file always produced a complete line. And the next write action would start on a new line.

Since Fortran 2003 the language also supports *stream-access* for both unformatted and formatted files. Before that standard, there were several more or less popular extensions to achieve the same effect.

8.1 Standard input and output

FORTRAN had no concept of standard input and standard output files. But there was a de facto standard in the form of logical unit numbers that were predefined:

- The unit number `6` was typically connected to the terminal:

```
      WRITE(6,*) 'Enter a number:'
```

- As an alternative you can also use the `PRINT` statement: it always writes to standard ouput:

```
*     Write to the terminal ...
      PRINT(*) 'Current value: ', value
```

- The unit number `5` was typically connected to the keyboard:

```
*     Let the user type a number and read it
      READ(5,*) value
```

- Other "magic" unit numbers could exist, such as `7`, often connected to a tape drive.

- Standard error could be connected to unit number `0` on some systems and would behave differently than standard output: no buffering, not redirected to file without extra care.

The FORTRAN 77 standard introduced the asterisk as a convenient way to define standard input and output without referring to a specific number: even though almost universal, these numbers were not defined by the FORTRAN standard.

With the advent of Fortran 2008 the intrinsic module `iso_fortran_env` now defines parameters like `INPUT_UNIT` to indicate the logical unit number for standard input and similar ones for standard output and error.

8.1.1 Carriage control

Printers of old had a simple system to control the positioning of the output on the paper: a character in the first column of a line to be output determined what would happen. Here are the (common) codes:

Character	Meaning
space	Go to the next line – the usual way of positioning
0	Skip one line
1	Go to the next page
+	Do *not* go to the next line – print over the current

On some systems it would work on the terminal as well.

An extension to the write statement that had to do with controlling the cursor on a terminal was the dollar sign:

```
WRITE(6,'(a,$)') 'Enter a number:'
```

The dollar sign meant that the cursor should stay on the last written line.

8.2 Direct-access files

By default, direct-access files are unformatted – values are dumped to the file or retrieved without the help of a human-readable format. Properties of direct-access files:

- You have to specify a record length, which holds for all records, when opening such a file. You can, however, open the file with different record lengths, if your application benefits from that. In other words, the length is not a property of the file itself.
- Direct-access files can be read or written by specifying a record number. Thus, like the name implies, you can jump around in the file at will.
- The record length is usually specified in *bytes* but some compilers, like Intel Fortran, use *words* as the unit. Where there was no way to know programmatically in FORTRAN what size was meant, since Fortran 2003, the intrinsic module `iso_fortran_env` contains the parameter `file_storage_unit` which is the size in *bits*.

Direct-access files, due to their simplicity, are compact and portable, as the structure of unformatted sequential files depends on the compiler that was used for building the program (see below).

The main issues that make these files non-portable are the binary representation of the numbers they contain. Nowadays, the main variation is the *endianness*: the order of the bytes that make up the number.

Here is a simple example of opening, writing and reading a direct-access file:

```
      PROGRAM DIRECTACCESS

      REAL VALUE(10)
      INTEGER I, REC

      OPEN( 10, FILE = 'directaccess.bin', ACCESS = 'DIRECT',
     &      RECL = 4*10 )

*
* CALCULATE SOME DATA AND WRITE THEM TO THE FILE
*
      DO 120 REC = 1,10
          DO 110 I = 1,10
              VALUE(I) = I + REC * 10.0
  110     CONTINUE

          WRITE( 10, REC = REC ) VALUE
  120 CONTINUE

*
* READ THE DATA - IN REVERSE ORDER
*
      DO 220 REC = 10,1,-1
          READ( 10, REC = REC ) VALUE(1), VALUE(2)
          WRITE( *, * ) VALUE(1), VALUE(2)
  220 CONTINUE
      END PROGRAM
```

It produces output like:

```
 101.000000       102.000000
 91.0000000       92.0000000
 81.0000000       82.0000000
 71.0000000       72.0000000
 61.0000000       62.0000000
 51.0000000       52.0000000
 41.0000000       42.0000000
 31.0000000       32.0000000
 21.0000000       22.0000000
 11.0000000       12.0000000
```

8.3 Unformatted sequential files

Data in sequential files, as the name suggests, are accessed in the order in which they appear in the files. For unformatted files you write the records one by one and you can read the records back one by one. But you cannot read more data from the record than its length. This is actually encoded in the file. The

following program will therefore fail, as it tries to read more data than present in the first record:

```
      PROGRAM SEQUNFORM

      REAL VALUE(10)
      INTEGER I, J

      OPEN( 10, FILE = 'sequnform.bin', FORM = 'UNFORMATTED' )

*
* CALCULATE SOME DATA AND WRITE THEM TO THE FILE
* THE RECORDS GET LONGER
*
      DO 120 J = 1,10
          DO 110 I = 1,10
              VALUE(I) = I + J * 10.0
  110     CONTINUE

          WRITE( 10 ) (VALUE(I), I = 1,J )
  120 CONTINUE

*
* GOTO THE START AND THEN
* READ THE DATA - IN REVERSE ORDER
*
      REWIND( 10 )

*
* FIRST RECORD: ONLY ONE VALUE
*
      READ( 10 ) VALUE(1)
      WRITE( *, * ) VALUE(1)

*
* SECOND RECORD: TWO VALUES, BUT READ TEN
*
      READ( 10 ) VALUE
      WRITE( *, * ) VALUE(1)

      END PROGRAM
```

Reading the first record works, but it fails on the second, as the program tries to read 10 values (the whole array!), whereas the record only contain two (the gfortran compiler was used):

```
   11.0000000
```

```
At line 38 of file sequnform.f (unit = 10, file = 'sequnform.bin')
Fortran runtime error: I/O past end of record on unformatted file

Error termination. Backtrace:

Could not print backtrace: libbacktrace could not find executable to open
#0  0xa11301fa
#1  0xa11278a1
#2  0xa1122d90
#3  0xa1148e3e
#4  0xa1130e81
#5  0xa1101841
#6  0xa11018e4
#7  0xa11013bd
#8  0xa11014f5
#9  0xb56c7613
#10  0xb76a26a0
#11  0xffffffff
```

One essential difference between unformatted sequential files and stream-access files, which were introduced in the Fortran 2003 standard, is that these unformatted sequential files hold extra bytes to indicate the records. These extra bytes are an implementation detail, not defined in the standard. A common scheme is:

- The record starts with an integer number of four bytes that defines how many bytes will follow.
- The bytes after that integer are the actual data.
- The record is closed with the same integer number, so that a check is possible but it is also possible to go back in the file via the `BACKSPACE` statement[12]

Thus the following statement has the effect of skipping a record:

```
READ(10)
```

whereas with a stream-access file nothing would happen, as these files have no concept of a record.

A common extension was that of the so-called *binary* file. Such a file is essentially what we now know as stream-access, but opening a file as *binary* required a specific, compiler-dependent keyword and not all compilers supported this type. Here is an example:

```
OPEN(10, FILE = 'data.bin', FORM = 'BINARY')
```

[12]The BACKSPACE statement originated with tapes as main mass data storage.

8.4 Reading an array

FORTRAN 77 defined a convenient way to read in arrays:

```
REAL A(100)
...
READ(10,*) A
```

This reads 100 numbers from the file connected to unit number `10`. But you can also use the following method – a so-called implied do-loop:

```
INTEGER I, N
REAL A(100)
...
READ(10,*) N
READ(10,*) (A(I), I = 1,N)
```

A clever trick is to read the number of values and the values themselves in one statement:

```
INTEGER I, N
REAL A(100)
...
READ(10,*) N, (A(I), I = 1,N)
```

Because the value of N is known when the do-loop starts, there is no problem reading the array in this fashion. (Unless, of course, N exceeds the size of the array.)

9 Some caveats for programmers coming from C-like languages

Some features of FORTRAN and indeed Fortran may confuse programmers who are used to C-like languages. This section is meant to at least point out a few such features. It is evidently far from complete. Since many of these features are common to both FORTRAN and Fortran, I will use the word Fortran in this section to mean both versions.

9.1 Terminology and semantics

Fortran uses the term *argument* to indicate the variables that are passed to and from subprograms, where C uses the term *parameter*. A *parameter* in Fortran, however, is a named constant. In a debugger it is most often not visible at all.

Another thing to watch out for is that a *pointer* in Fortran is not merely a memory address, as it is in C. A Fortran pointer holds far more information and it can only point to variables that are either themselves pointers or have the *target* attribute.

In C assignment, `a = b`, is actually an operator. In Fortran, it is a statement and therefore cannot be put into a combination like: `a = b = c`.

9.2 Subroutines and functions

A function in Fortran always returns a value and you cannot ignore that, like in C. If `f` is a function (either in C or Fortran) returning an integer value, then:

```
f( x ); /* Ignore the return value */
```

is valid C, but in Fortran the return value always has to be handled.

The default method for passing variables to subroutines or functions in Fortran is *by reference*, whereas in C it is *by value*. That means, unless you specify the intent to be `intent(in)`, then you can always change the value of the actual variable.[13]

9.3 Initialisation of variables

In C code like:

```
int f( int x ) {
    int y = 1;

    if ( x > 0 ) {
        y = 0;
    }
    return x + y;
}
```

is simply a short-hand for:

```
int f( int x ) {
    int y ;

    y = 1;
    if ( x > 0 ) {
        y = 0;
    }
    return x + y;
}
```

that is, the variable `y` is set to 1 at the start of the function every time it is called.

The Fortran code that *looks* like exactly this:

```
integer function f( x ) {
    integer, intent(in) :: x

    integer y = 1
```

[13] Intent did not exist in FORTRAN.

```
    if ( x > 0 ) then
        y = 0
    endif
    f = x + y
end function f
```

behaves in a different way:

- The local variable `y` has the value 1 at the start of the program and implicitly has the `save` attribute, so it retains its value.
- It is *not* reset to 1, unlike in the C version.
- If an argument `x` is passed with a positive value, then the variable `y` is reset to 0 and keeps that value from there on. This is a side effect!

In fact the Fortran code is equivalent to:

```
int f( int x ) {
    static int y = 1;

    if ( x > 0 ) {
        y = 0;
    }
    return x + y;
}
```

9.4 Handling character strings

C does not truly have character strings, it has arrays of characters. As C does not have intrinsic features to keep track of the size of an array, it uses a convention to make sure that strings are terminated – the trailing NUL byte. A C program is itself responsible for maintaining this NUL byte.

In contrast, Fortran uses strings of a defined length and pads strings with spaces. Thus:

```
character(len=10) :: a

a = '123'
```

means that the `a` has the value `'123_______'` (where `'_'` is used to indicate a space).

This makes programming with strings a lot easier as you can simply rely on the compiler to make sure that strings fit into the allotted memory:

```
character(len=10) :: a, b
character(len=15) :: c

a = '1234567890'
```

```
b = 'abcdefghij'

c = a // b

write(*,*) c
```

will print the string `'1234567890abcde'`, without any concern for the fact that `a // b` is longer than the variable `c` can hold.

10 Subjects

```
- array(*) versus array(:)
- history of computers:
   - hardware
   - memory management
   - tools like source code control systems
   - connections between computers, Internet
- command-line arguments for file names
- numerical binary representations versus IEEE (IBM, Cray, Convex)
- list-directed input and output - also: /
- narrow formats (?)
- use of d00 in input
- Cray pointers
- external, also notation "*tan"
- FORTRAN 66 semantics: OPEN - STATUS = 'NEW' as default.
- Effect of BLANK = 'ZERO' versus BLANK = 'NULL'
- Equating logicals: logical x, y; if ( x .eq. y ) then
- r1mach and friends
- Declaring an argument to be a scalar but using it as an array
- character*1 a*2
```

Done:

```
- intent
- checking interfaces
- equivalence
- big-endian and little-endian
- double complex
- double precision versus kind
- temporary arrays - non-contiguous arrays
- six characters
- constants as actual arguments
- fixed form and spaces
- array(10) as the starting point
- implicit types
- separate compilations, the misunderstanding of one routine per file
```

```
- LU-numbers 5 and 6 (and 7)
- standard input and output
- real do-variables
- entry
- statement functions
- unformatted versus binary files
- uppercase/lowercase letters
- D as comment character
- Specific names for functions like max() and sin()
- Alternate return
- Hollerith
- DECODE/ENCODE:

From discourse: (19 june 2023)
   CHARACTER S*6 / '987654' /, T*6
   INTEGER V(3)*4
   DECODE( 6, '(3I2)', S ) V
   WRITE( *, '(3I3)') V
   ENCODE( 6, '(3I2)', T ) V(3), V(2), V(1)
   PRINT *, T
   END

The above program has this output:

98 76 54
547698

The DECODE reads the characters of S as 3 integers, and stores them into V(1), V(2), and V(3
The ENCODE statement writes the values V(3), V(2), and V(1) into T as characters; T then con
```

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