Modernizing Old Fortran

Modernizing Old Fortran

Many Fortran programs written in the early years are of continuing value, but old source code does not always work unaltered with modern compilers. Where the compiler produces error messages, these notes may help in finding ways to fix the offending statements.

In theory there should be few problems because the designers of Fortran have always taken trouble to retain compatibility with older forms of the language. Each successive Fortran Standard has incorporated the preceding Standard as a subset, with only a few exceptions. In the few cases where a feature has been dropped it is because there is a superior replacement and the original was seriously problematic. Some dropped features were completely unportable, for example Hollerith constants and the storage of character strings in numerical variables was dropped when the CHARACTER data type was introduced in Fortran 77. Others were dropped because they were wholly incompatible with the principles of structured programming, for example the ASSIGN statement and assigned GO TO.

Changes are sometimes appropriate because old code written in Fortran 77 or Fortran 66 does not actually conform to either official standard. Unfortunately many of the recent changes come from coders who are less experienced, so are not aware that they are adding unnecessary compilations. Compiler-specific extensions were often used, for a variety of reasons:

- Many programmers were scientists or engineers who were more concerned with code that worked on limited computers than language niceties.
- Limitations in earlier forms of Fortran meant that some desirable features, such as dynamic storage allocation, could not be used without resorting to extensions.
- Some code was written for a specific platform, so standard-conformance was of less importance.

Note: reference to DEC Fortran in what follows are to the series of compilers produced by the Digital Equipment Company for its PDP-8, PDP-11, and VAX series of computers. The company was subsequently taken over by HP, but the Fortran division was absorbed into Intel Fortran, whose compilers are most likely to contain appropriate compatibility features.

ACCEPT Statement

Reads from standard input. Example:

```
      ACCEPT *, X, Y, Z
```

Status: in DEC Fortran, always non-standard.

Action: read from standard input using * as the I/O unit, e.g.

```
read *, x, y, z
```

Alternate RETURN

This provides a way, when returning from a CALL statement, to jump to some other labelled point in the calling routine, typically in the event of some error condition. Example:

```
      CALL MYSUB(args, *123, *456)
C ... code here for normal return
```

```
123   CONTINUE
C ... code here for to handle abnormal exit
456   CONTINUE
C ...  ditto for some other condition

      SUBROUTINE MYSUB(args, *2, *3)
C *N corresponds to a RETURN statement with given integer value
      IF(ENDFIL) RETURN 2
      IF(ERROR) RETURN 3
```

Status: Declared obsolescent in Fortran 90, but still standard.

Action: still standard-conforming so can be left unchanged at present. A better solution is to change the code to return an integer argument and test the value in the calling routine after the subroutine call e.g. using an if-block or select case structure.

Arithmetic IF

A three-way branch to labelled statements depending on the value of a real or integer expression: less than zero, equal to zero, greater than zero. Two of the labels may be the same for a two-way branch. Example:

```
      IF(X-Y) 10, 20, 30
10    CALL YBIG
      GO TO 100
20    CALL EQUAL
      GO TO 100
30    CALL XBIG
```

Status: Declared obsolescent in Fortran 90, but still standard.

Action: may be left unchanged. Better to use a SELECT CASE block with integer expressions, or a IF/ELSEIF/ELSE/ENDIF block for real expressions to avoid the clutter of statement labels.

ASSIGN Statement and assigned GO TO

This allows the value of a statement label to be assigned to an integer variable, which can be used in a subsequent GO TO statement to procure a jump to one of a preset number of labelled statements. This seems to have been a feature of the very earliest Fortran in order to allow internal subroutines and functions, where one needs a return to a statement just after the calling point. The GOSUB statement in the Basic language is a more elegant way of doing the same thing. Example:

```
      ASSIGN 1234 TO MYPOS
      GO TO 1000
1234  CONTINUE
C ------ INTERNAL FUNCTION: HAVERSINE -----
1000  Y = 2.0 * (SIN(X/2))**2
      GO TO MYPOS
```

Status: In Fortran 90 declared obsolescent, deleted in Fortran 95 (but still supported by some current compilers as an extension).

Action: rarely encountered in the wild, but needs careful handling. If used as demonstrated above it can be replaced by an internal subroutine or function, but some programmers may have played more elaborate tricks with ASSIGN which need careful analysis.

Assigned FORMAT

Allows one of a number of format statements to be selected in advance to be used with a given read/write statement. Example:

```
      ASSIGN 900 TO MYFMT1
      WRITE(6,MYFMT1) RESULT
      ASSIGN 901 TO MYFMT2
      WRITE(6,MYFMT1) RESULT
C...
```

```
900   FORMAT(F10.0)
901   FORMAT(E10.1)
```

Status: In Fortran 90 declared obsolescent, deleted in Fortran 95 (but still supported by some current compilers as an extension).

Action: usually simplest to store the complete format string in a character variable; if this is given the same name as the integer used originally, then the read and write statements do not need to be changed.

```
character(len=10) :: myfmt1
myfmt1 = '(f10.0)'
write(*,myfmt1) result
myfmt1 = '(e10.1)'
write(*,myfmt1) result
```

Assumed-length Character Functions

The actual length of the function is that declared for the name in the calling routine. Example:

```
      CHARACTER UPCASE*10, STRING*20
C...
      STRING = UPCASE('test')
C...
      CHARACTER*(*) UPCASE(ARG)
```

This does not do what it appears: the function always returns a string of 10 characters when called from this routine because of the declaration (but references to the function from other program units may return different lengths).

Status: declared obsolescent in Fortran 95, because it is misleading, and the functionality that it does not quite provide can be done properly other ways, such as a subroutine with passed-length arguments, or a function with a return argument with a length which does depend upon its arguments.

Action: if it really works best to leave it alone, but note it as generally the mark of a programmer being too clever by half.

Assumed-size Arrays

Example:

```
      SUBROUTINE SUB(IMAGE)
      REAL IMAGE(512,*)
```

Status: declared obsolescent in Fortran 95[1], because it can only be used for the **last** dimension of a multi-dimensional array, and provides no information on the actual upper-bound to the called procedure.

Action: If the procedure has an explicit interface, for example it is (or can be) in a module, then use an assumed-shape array (colon replaces the asterisk).

It should be noted that there are some tricks that you can do with assumed-size arrays that you can’t do as easily with assumed-shape arrays. One example, that is relatively common in linear algebra routines, is to pass sub-matrices by passing the top left hand element, i.e. with

```
      SUBROUTINE SUB(A, LDA)
      REAL A(LDA, *)
```

then

```
      CALL SUB(A,LDA)
```

passes the full matrix and

```
      CALL SUB(A(3,2), LDA)
```

passes a sub-matrix. Whilst it is possible to do this with assumed-shape arrays, by slicing, this often results in the compiler creating internal copies of the data.

Branching to END IF from outside the block

A feature hardly ever used in practice as it has no value, but something the earlier standards forgot to prohibit.

Status: Declared obsolescent in Fortran 90, deleted from Fortran 95.

Action: Can easily be replaced by a jump to e.g. a labelled CONTINUE statement after the END IF.

CALL or function call with omitted arguments

Optional arguments were only standardized in Fortran 90, but many earlier compilers implemented them and allowed pairs of commas in CALL statements or function references to denote omitted arguments. For example:

```
      CALL MYSUB(x,,z)
```

Status: Never standardized, common extension in DEC Fortran and others.

Action: Reorder the arguments so that the optional ones are at the end of the list; check that the formal arguments are declared optional within the procedure.

CHARACTER✱length

Example:

```
      CHARACTER*10 STRING
```

Status: Declared obsolescent in Fortran 95.

Action: Can be replaced with the new syntax such as CHARACTER(LEN=10) which is clearer and more consistent with other forms of data type declaration.

Carriage-control Convention

For many years output to a device called a “printer” was treated specially in Fortran to provide a primitive way of controlling page layout. The first character of each line was removed and used to control the line and page spacing:

character	Effect
space	Output on next line
0	Skip one line
1	Skip to next page
+	Output continues on same line

It should be noted that the effect of “+” as first character was especially device-dependent: on some devices it would continue the same line, on others would overwrite starting at the left margin again, on others it would appear to have no effect. Although this technically applied only to output destined for a “printer”, many compilers especially on Unix-like operating systems, could not tell whether output would go to a printer or not, so applied it to all output text files, which could be very annoying. The practice appears to have died out in recent years, but lives on in the specification for list-directed output, which requires a leading space at the start of every output line. Its influence in old code can be recognised by the presence of something like “1X” or even “1H ” at the start of each output format.

Status: Deleted from Fortran 2008 (without any period of being “obsolescent”, but not before time, many think).

Action: None needed in general. Blank lines can easily be produced by using slashes in output formats. There is no general way of getting a page throw on modern printers, except by using a specific printer control language. Those writing new code should note that they no longer need to worry about the first character on each line.

Computed GO TO

Branches to one of a number of labelled statements according to the value, 1 to N, of an integer variable. Example:

```
      GO TO (101, 102, 103), MYVAL
101   X = X + Y
      GO TO 200
102   X = X - Y
      GO TO 200
103   X = X * Y
200   CONTINUE
```

Status: Declared obsolescent in Fortran 95.

Action: May be replaced by the SELECT CASE structure, which is more powerful and does not involve the risk of failing to include a GO TO statement in each clause to branch to the end. It also allows a **`case default`** section as a long-stop. For example:

```
select case(myval)
case(1)
   x = x + y
case(2)
   x = x - y
case(3)
   x = x * y
case default
   print *,'Invalid value of myval'
end select
```

DATA Statement among Executable Statements

Early forms of Fortran allowed the DATA statement, uniquely among non-executable statements, to be intermixed with executable statements. There seems little point in doing this.

Status: Declared obsolescent in Fortran 95.

Action: Simply move the offending DATA statements to a position above all the executable statements in that program unit. In some cases the initialisation of a variable can be done more clearly in the corresponding type declaration statement.

DECODE Statement

An early form of what later became the internal file READ, reading a string of characters under format control to extract (usually) numerical values. Example:

```
      DECODE(20, 105, string) areal, breal
105   FORMAT(F10.2, F10.4)
```

The first item in the list is a character count, no longer needed in an internal file read. Optionally the items in parentheses could include IOSTAT=intvar and ERR=label.

Status: Never standardized, but in DEC Fortran and many others.

Action: Convert to an internal file read, e.g.

```
read(string, '(f10.2, f10.4)') areal, breal
```

DEFINE FILE Statement

Originally an IBM Fortran statement to specify the properties of a direct-access file. Example:

```
      DEFINE FILE unit(nrecs, lenrec, U, ivar)
```

This defines a file consisting of exactly nrecs records each of length lenrec - but the units of this were system-dependent, 16-bit records on early systems, 32-bit records (perhaps) on later ones. The U argument was required to denote an unformatted file, the final integer variable was the **associated variable** which was updated with (bizarrely) the number of the record **after** the record just read/written. DEFINE FILE did not actually open the file: this was done by the first READ or WRITE accessing that unit.

Status: Never standardized.

Action: Replace by an OPEN statement using ACCESS="DIRECT" and a suitable RECLEN value (note units are still system-dependent, but an option of the INQUIRE statement allows you to find the appropriate length). Any programs which actually make use of the associated variable need careful examination as there is no simple equivalent in standard Fortran.

DIMENSION size (1)

An early attempt to provide array arguments of unspecified length. Example:

```
      SUBROUTINE APROC(ARRAY)
      DIMENSION ARRAY(1)
```

Status: commonly used in Fortran 66 at a time when compilers did not check array-bounds against the declared length, so the array size of the actual argument could be flexible. The feature was deleted in Fortran 77, but may still work with current compilers if array-bound checking is not enabled.

Action: the simplest replacement is to replace the 1 by an asterisk, to make it an assumed size array, but this was declared obsolescent in Fortran 95. A better solution is to use a colon, turning it into an assumed shape array, but this requires an explicit interface.

DO with transfer into range from outside it

This was theoretically permitted in Fortran 66 but little used, fortunately.

Status: Removed in Fortran 77.

Action: If this is encountered in the wild, the logic of the program needs to be examined carefully to see why the jump out and back in is done, and then the program flow refactored appropriately.

DO with real loop index

Example:

```
      REAL X
      DO 15, X = 0.5, 5.0, 0.5
```

Status: Declared obsolescent in Fortran 90, deleted from Fortran 95, because the repeated addition of the increment combined with the approximations inherent in floating point arithmetic made the actual number of loops somewhat processor-dependent. Many compilers continue to permit this, as an extension, for compatibility with old code.

Action: Should be replaced by a DO loop with an integer loop variable, and then an assignment to a real variable, providing you can work out how many loops the original programmer intended. For example:

```
do ix = 1,10
   x = 0.5 * ix
```

DO with shared termination

Example:

```
      DO 100 J = 1,NJ
      DO 100 I = 1,NI
      X = X + ARRAY(I,J)
100   CONTINUE
```

Status: Declared obsolescent in Fortran 90, but still part of the Standard.

Action: Better to replace the labelled form of DO with DO - END DO throughout, and indent the contents, which solves the problem much more neatly.

DOUBLE COMPLEX statement

Status: The complex data type only had one precision (single) in Fortran 77, but this was a fairly common extension.

Action: Replace by a COMPLEX statement using a suitable kind parameter. For example:

```
integer, parameter :: dp = selected_real_kind(15,50)
complex(dp) :: variable, list
```

DOUBLE PRECISION statement

Status: In Fortran 77, double precision was one of the six intrinsic data types, but since Fortran 90 it has been relegated to being an alias for the higher precision kind of the **real** type.

Action: None required, but ideally new code should make use of the SELECTED_REAL_KIND function to select the kind of a floating point variable according to the range and precision required, as this is more portable. The DOUBLE COMPLEX statement is sometimes encountered: in modern Fortran both **real** and **complex** data types have a higher precision kind, so this does require modification to be standard-conforming. Note that double precision constants using the “d” suffix are also obsolescent, as appending an underscore and the appropriate kind value is more portable.

ENCODE Statement

An early form of what later became the internal file WRITE, converting usually a numerical variable to the corresponding string of characters under format control. Example:

```
      ENCODE(20, 105, string) xreal
105   FORMAT(F10.2)
```

The first item in the list is a character count, no longer needed in an internal file write. Optionally the items in parentheses could include IOSTAT=intvar and ERR=label.

Status: Never standardized, but in DEC Fortran and many others.

Action: Convert to an internal file write, e.g.

```
write(string, '(f10.2)') xreal
```

ENTRY Statement

The ENTRY allowed a function or subroutine to have two or more entry points, which could have different sets of formal arguments. This is incompatible with the best principles of structured programming, and leads to programs which are hard to understand.

Status: Declared obsolescent in Fortran 2008.

Action: Simplest to leave this alone. In principle a better structure can generally be obtained using either a single entry point with optional arguments, or else several thin shells with individual argument lists, all of which call the same lower-level routine. In either case a lot of refactoring is required.

EQUIVALENCE Statement

The EQUIVALENCE statement is still part of modern Fortran but when found in old code it often turns out to be used in ways which are non-standard, for example to associate numerical variables with ones of character type, or to transfer bit-patterns from one data type to another. The TRANSFER intrinsic function should be used to do this. One form of EQUIVALENCE which conformed to Fortran 66 but not later standards was to have a reference to a multi-dimensional array with just one subscript.

FORMAT descriptors, obsolete

nH descriptor

Status: The nH descriptor was declared obsolescent in Fortran 90 and removed from Fortran 95. In early Fortran nH was also permitted for input but this was removed in Fortran 77.

Action: In output formats it can easily be replaced by a character string descriptor e.g. 4Hthis becomes “this” or ‘this’. If encountered in used with READ the code needs careful inspection, since it had the effect of altering the FORMAT string e.g. to update a column header, for subsequent output. Reading into a character variable might be a suitable replacement.

Q descriptor

Status: The Q descriptor has never been standard, but was used in DEC Fortran and others with READ statements to transfer the length (or remaining length) of an input text line to be transferred to an integer variable.

Action: Modern Fortran has no exact equivalent, but a non-advancing READ can use the `size` keyword to get similar functionality.

$ and \ descriptors

Status: Never standard, but $ was used in DEC Fortran and many others to suppress the end-of-line sequence (such as CR and LF) at the end of a line of text. “\” was used in a few other compilers for the same purpose.

Action: Replace with a non-advancing write statement.

Fixed-form Source Code

Status: Declared obsolescent in Fortran 95, but still part of Fortran 2008.

Action: No need to change, but the modern free-format layout has many advantages, and there are some free converters to this form, including Michael Metcalfe’s convert.f90 and Alan Miller’s to_f90.f90 .

Functions, library

Hollerith Data and Constants

Status: Hollerith constants in expressions (as opposed to within Format specifications) were removed in Fortran 77 when the character data type was introduced. A few compilers continued to allow them as an extension.

Action: Replace with a character string enclosed in quotes or apostrophes.

I/O Units 5, 6, and 7

Some early IBM computers with Fortran compilers pre-connected certain input/output units to devices, typically:

Unit	Device
5	Input from card reader
6	Output to line printer
7	Output to card punch

Other Fortran compilers tended to follow suit, and 5 is still pre-connected for input from the user's keyboard, and 6 for output to the user's screen, even on modern systems.

Status: Never defined in any Standard.

Action: Best to replace the unit number with an asterisk, which is a standard notation for reading from the standard input unit or writing to standard output unit. Card punches seem to have vanished from the scene.

Note: I/O unit 0 is sometimes seen in Fortran designed for use on Unix systems and similar: on many compilers it was pre-connected to the standard error output stream and was meant to separate error messages from regular text output. In Fortran 2008 the intrinsic module ISO_FORTRAN_ENV has an item ERROR_UNIT which has the same purpose.

Initial values in type statements

Example:

```
      REAL X / 3.14159 /, ANSWER / 42.0 /
```

Status: Never standardized, but a common extension in early Fortrans.

Action: Replace with modern syntax, e.g.

```
real :: x = 3.14159, answer = 42.0
```

Note that the double colon is needed to allow initial values to be specified.

OPEN/CLOSE with non-standard keywords

In DEC Fortran OPEN statements many non-standard keywords may appear; a few which have simple translations into modern Fortran are shown in this table.

DEC Fortran keyword	Modern Equivalent
ACCESS = ‘APPEND’	position = ‘append’
NAME = ‘filename’	file = ‘filename’
READONLY	action = ‘read’
RECORDSIZE = n	recl = n
TYPE = ‘OLD’	status = ‘old’ (etc.)

The DEC Fortran CLOSE statement might also use “DISPOSE=” which can be replaced by “STATUS=”.

PARAMETER Statement (DEC Fortran form)

Example:

```
      PARAMETER name = 12345.6
```

Status: This simple form was in DEC Fortran and perhaps others, but never standardized.

Action: Simplest just to put parentheses around the name=value pairs, but better to absorb the statement into a type statement with a parameter attribute, e.g.

```
real, parameter :: name = 12345.6
```

PAUSE Statement

This caused execution to pause, after which it could be resumed by some action by the computer operator, e.g. after a tape had been mounted.

Status: Declared obsolescent in Fortran 90, deleted from Fortran 95.

Action: Best to replace by a write statement sending the message (if any) to the standard output unit, followed by a read from the standard input unit.

Pre-connected I/O Units

Before the OPEN statement was introduced in Fortran 77, there was no standard way of associating I/O unit numbers with external files: this was left to job control. Typically the first time a program executed a WRITE statement on a given unit (provided the number was not 5, 6, or perhaps 7) the run-time system would create a file with a suitable name. For example if you write to unit 13 then the file might be called “FOR013.DAT” or perhaps “fort.13” but systems varied a lot in the names they used. Some systems would allow automatic connection of input files provided they already existed with a name of the required form.

Status: A feature not covered by any Fortran Standard.

Action: Use an OPEN statement to connect each I/O unit with a named file.

REAL*8 etc.

Examples:

```
      REAL*8 dblex, bigone
      INTEGER*2 shorty
```

status: The notation mirrors that of the CHARACTER statement, but for other data types has never been permitted by any Fortran Standard. Despite this, many compilers support it. The notation is designed to specify the number of bytes used to store the variables: thus `REAL*8` specifies 8 bytes, often corresponding to double precision type. Although the notation is simple and has a fairly obvious meaning, Fortran has never been restricted to byte-oriented hardware. The designers of Fortran 90 designed what they thought was a superior replacement involving variables of different kinds, where the kind can be chosen for the required values of precision and exponent range.

Action: `REAL*8` and `INTEGER*2` can be replaced by something like this:

```
integer, parameter :: dp = selected_real_kind(15, 50)
real(dp) :: dblex, bigone
integer, parameter :: short = selected_int_kind(3)
integer(short) :: shorty
```

Note that there is no guarantee that this integer specification will result in 2-byte (16-bit) wordlength, only that it will be enough to store numbers of 3 decimal digits. If the hardware has no suitable number representation of this size, then variables with a longer wordlength will result.

STRUCTURE, RECORD, MAP, UNION Statements

Status: These originated in DEC Fortran as a way of declaring data structures. The derived-type definition in Fortran 90 replaced the STRUCTURE and RECORD statements but is different in many details. There is no standard equivalent of MAP and UNION statements which were designed to allow aliases for elements of data structures.

Action: STRUCTURE and RECORD can be converted to modern standard forms in a straightforward fashion by using derived types, but some work is likely to be involved. Replace the “STRUCTURE /structure-name/” statements with “TYPE structure-name” statements, and replace “RECORD /structure-name/” statements with “TYPE(structure-name)” statements.

```
type person_t  ! was: structure /person_t/
  character(64) :: name
  integer :: age
end type       ! was: end structure

type(person_t) :: people_array(100)  ! was: record /person_t/ peopl
```

Access to derived type components is with the percent character ("%"), rather than the period ("."):

```
  do, i=1, size (people_array)
    print *, i, trim (people_array(i)%name), ', ', people_array(i)%
  end do
```

If UNION or MAP statements were used, the difficulties are much larger. Use of Fortran 2003 type extension (EXTENDS) and polymorphism (CLASS) may be of value. For example, using type extension different specializations could be defined:

```
  type, extends(person_t) :: student_t  ! was a map of a union
    integer :: student_id
    real :: gpa
  end type

  type, extends(person_t) :: employee_t  ! was a map of a union
    integer :: employee_id
    character(:), allocatable :: title
    real :: salary
  end type
```

By using CLASS and SELECT TYPE, procedures may be written which generically support type person_t and any of its specializations:

```
  subroutine person_print (this)
    class(person_t), intent(in) :: this
    class(person_t), intent(in) :: this

    print *, 'name, age: ', this%name, ', ', this%age
    select type (this)
    type is (student_t)
      print *, 'student ID and GPA: ', this%student_id, this%gpa
    type is (employee_t)
      print *, 'title: ', this%title
      print *, 'employee ID and salary: ', this%employee_id, ', $'
    end select
  end subroutine
```

Statement Functions

Status: These are single statements each of which defines a function for use within the same program unit. They were declared obsolescent in Fortran 95.

Action: Still standard Fortran so can be left alone, but a better replacement is to define internal functions (or subroutines) where the body is not limited to what can be achieved in a single expression.

TYPE Statement (DEC Fortran)

Example:

```
      TYPE "The value of X is", X
```

Status: Never standard, only in early DEC Fortran.

Action: replace with a PRINT statement, or WRITE statement using the standard output unit.

Tab Characters in Source Code

Status: Standard Fortran has never accepted tab characters in source-code, but many compilers permitted them, but treated them in different ways. Since their presence is normally invisible they are best avoided.

Action: use a text editor or some other utility to translate them to a suitable number of spaces. In modern Fortran using free-format layout translating them to a single space might suffice.

WRITE(unit'record)

Status: A DEC Fortran notation for writing direct-access records.

Action: Change to `WRITE(unit, rec=record)`.

%VAL, %REF, etc

Status: these are completely non-standard functions used in passing actual arguments by different mechanisms to the usual, typically to obtain dynamic storage in the age of Fortran 77, or to interface with procedures written in C or other languages.

Action: there is no standard alternative, but most of the features that these functions supported have simpler equivalents in modern Fortran, for example: dynamic memory, and (from Fortran 2003 on) a standard way of interfacing with C-language code.

Comments

The recent version of this article contained many quite arrogant and incorrect statements, many of which have been removed. The reality is that code such as Lapack and many other scientific codes were written to work on limited computers, and "niceties" of standards was less of an issue. Sadly many recent changes are pedantic rather than pragmatic, making writing Fortran harder not simpler. So it goes; two steps forward, one step back – or perhaps three steps back.

---

1. Assumed size arrays still appear to be "current" in F95 and subsequent revisions of the standard, based on the font used to describe them and

their omission from the list of obsolescent features in Annex B of each standard. ↩