Introduction to Modern Fortran

More About I/O and Files

Nick Maclaren

nmm1@cam.ac.uk

March 2014

Summary

The features here are important for real code

- You don’t need to know them in detail
- You need to know where “gotchas” occur
- You need to know what Fortran can do for you

So you don’t waste time reinventing the wheel

Writing Buffers etc.

All files are closed at program termination
All unwritten output will be written to disk

- It does not happen if the program crashes

It is a good idea to close files yourself
Or to force the output to be written

- Especially for files containing diagnostics!

CLOSE

It's almost trivial:

```
CLOSE (1, IOSTAT=err)
```

You can delete a file as a CLOSE option

```
CLOSE (1, STATUS='delete', IOSTAT=err)
```

FLUSH

Fortran 2003 introduced a FLUSH statement
Can almost always clean up old code by changing to it

Causes pending output on a unit to be written
So a program crash doesn't lose output

```
FLUSH (99)
```

Older ones usually have a FLUSH subroutine
Argument usually just the unit number

```
CALL FLUSH (99)
```

ISO_FORTRAN_ENV (1)

An intrinsic (built–in) module ISO_FORTRAN_ENV

Specifies three non–negative integer constants:
INPUT_UNIT, OUTPUT_UNIT and ERROR_UNIT
Units corresponding to stdin, stdout and stderr

Negative constants IOSTAT_END and IOSTAT_EOR
Values set on end–of–file and end–of–record
Latter is not set for simple formatted READ
    Short records are simply padded with spaces

ISO_FORTRAN_ENV (2)

Unit numbers enable somewhat cleaner code
Don't need to use old UNIT=* form

IOSTAT values allow cleaner error handling

```
USE, INTRINSIC :: ISO_FORTRAN_ENV
INTEGER :: ioerr

READ (1, IOSTAT=ioerr) array
IF (ioerr == IOSTAT_END) THEN
   . . .
```

Testing for Connection

You can test if a unit is already connected
Can avoid using any preconnected ones by mistake

```
LOGICAL :: connected
INTEGER :: iostat
INQUIRE (UNIT=number, IOSTAT=iostat,    &
    OPENED=connected)
```

A non–zero value of iostat means unit is unavailable

More on Formats

Fortran formatted I/O is very powerful
But it is also complicated and messy

- Use only the facilities that you need
- If you start to write complicated code
  check for a built–in feature to do it

Even this will not mention all the features

Exponential Format (1)

En.m is the original leading zero form
ESn.m is standard scientific notation
ENn.m is what is called engineering notation

E:	0.0	$\leq$	|mantissa|	<	1.0
ES:	1.0	$\leq$	|mantissa|	<	10.0
EN:	1.0	$\leq$	|mantissa|	<	1000.0

EN displays an exponent that is a multiple of 3

Example

E??.3	ES??.3	EN??.3
0.988E+01	9.876E+00	9.876E+00
0.988E+02	9.876E+01	98.765E+00
0.988E+03	9.876E+02	987.654E+00
0.988E+04	9.876E+03	9.876E+03
0.988E+05	9.876E+04	98.765E+03
0.988E+06	9.876E+05	987.654E+03
0.988E+07	9.876E+06	9.876E+06
0.988E+08	9.876E+07	98.765E+06

Exponential Format (2)

The exponent is always exactly 4 characters
It depends on the value of the exponent

$\|exponent\| \leq 99$	$E\pm e_1 e_2$
$99 < \|exponent\| \leq 999$	$\pm e_1 e_2 e_3$
$999 < \|exponent\|$	field overflow

The last cannot occur for IEEE double precision
It can for IEEE quadruple precision and Intel

Setting the Exponent Width

You can set the exponent field width explicitly

En.mEk, ESn.mEk, ENn.mEk or Gn.mEk

k is the number of digits not the width

ESn.mE2 is similar to ESn.m, etc., but saner

E.g. WRITE (*, '(ES15.3E4)') 1.23D97

displays 1.230E+0097

Overflow of Exponent Field

Note what happens if the exponent is too large

	1.2d–5	1.2d–25	1.2d–125	1.2d–1250
ES9.1	1.2E–05	1.2E–25	1.2–125	******
ES9.1E1	1.2E–5	******	******	******
ES9.1E2	1.2E–05	1.2E–25	******	******
ES9.1E3	1.2E–005	1.2E–025	1.2E–125	******

Note that the overflow behaviour is saner
It's still rather user–hostile, unfortunately

Numeric Input (1)

F, E, ES, EN and D are similar
The valid number formats are identical

The n characters are decoded as a number
Spaces are ignored (even embedded ones)
A completely blank field delivers zero

- Any reasonable format is accepted

Plus a large number of very weird ones!
Unambiguous, because the field width is known

Numeric Input (2)

Good reasons for accepting weird formats
But they are now historical oddities

Warning: there are serious ‘‘gotchas’’ lurking
You may find that your input gets rescaled
That is multiplied or divided by a power of ten

I describe a bit of this in the extra, extra slides
The next one describes what to do to be safe

Numeric Input (3)

Follow any of these rules for REAL

- Use a descriptor like Fn.0 (e.g. F8.0)
- Always include a decimal point in the number
- Use a belt and braces – do both!

And don't use odd features not covered here

Example

Assume a format like F15.0 or F22.0
Any of the following inputs will produce 12.3

```
``   12.3            ''
``   1   2   .   3   ''
``   1.23e1          ''
``   +.123d+0002     ''
``0000000123.0e–1''
```

And so on

Reinput of Output

- Formatted I/O can reread anything it wrote
Unless the value was written as asterisks

Obviously, there may be some precision loss
Including any truncated CHARACTER data

- But it may not be readable in other ways
Not even via list–directed I/O or as code
E.g. 1.23–125 is not a valid REAL constant

A problem for only very big or small numbers

Other Descriptors (1)

SP and SS set and unset printing plus (+)
WRITE (*, '(SP, F8.3)') 2.34 displays +2.340

: halts if there are no more transfer list items
WRITE (*, '(I5, :, " cubits")') 123 displays “123”

T moves to an absolute position
TR is a more modern syntax for X

DT – used for derived types (Fortran 2003)

Other Descriptors (2)

DC and DP set comma versus decimal point

- P is historically essential and truly EVIL

Do NOT use it in an input format
OR if there are any F descriptors in the format
It will rescale values by a power of ten

Extremely esoteric and best avoided:

BN, BZ, RC, RD, RN, RP, RU, RZ, S, TL

Recycling of FORMATs

As mentioned, the transfer list is primary
Have described what happens if it is short
If it is long, the FORMAT is recycled

It starts a newline, as if there was a /
And restarts from the last parenthesised group
Which must contain at least one edit descriptor

'(F5.2, 5(I2, E12.3))' repeats '(5(I2, E12.3))'
'(F5.2, 5I2, 3E12.3)' repeats everything

Internal Files (1)

- These are CHARACTER variables or arrays

You can use them to convert to or from text
They are useful for creating dynamic formats

Each variable is a record of the same length

Arrays are a sequence of records
These are in array element order, as usual

Internal Files (2)

- Use the variable or array name as the unit
- Permitted ONLY for formatted I/O
- And only in READ and WRITE statements
- You can't use them for non–advancing I/O

There are a few other, obscure, restrictions

Example (1)

```
CHARACTER(LEN=25) :: buffer, input(10)
WRITE (buffer, '(f25.6)') value
IF (buffer(1:1) == '*') THEN
   buffer = 'Overflow'
ELSE
   buffer = TRIM(ADJUSTL(buffer)) // 'cm'
END IF
PRINT *, 'value=', buffer
```

Example (2)

```
READ (*, '(A)') input
DO k = 1,10
   IF (input(k)(1:1) /= '#') &
      READ (input(k), '(i25)') number
   . . .
```

Dynamic Formats (1)

Internal files are useful for for dynamic formats

- Yes, this example is easier in other ways

Let's say that we want the following:

CALL trivial ('fred', 12345)

To produce output like:

fred=12345

Dynamic Formats (2)

```
    SUBROUTINE trivial (name, value)
        CHARACTER(LEN=*) :: name
        INTEGER :: value
        CHARACTER(LEN=25) :: buffer1, buffer2

        WRITE (buffer1, '(I25)') value

        WRITE (buffer2, '("(A, ""="", I", I10, ")")') &
            26-SCAN(buffer1,'123456789')
!   WRITE (*,*) buffer2     ! to see the format it creates
        WRITE (*, buffer2) name, value

    END SUBROUTINE trivial
```

Dynamic Formats (4)

```
CALL trivial ('fred', 12345)
CALL trivial ('Jehosephat', 0)
CALL trivial ('X', 987654321)
```

produces:

```
fred=12345
Jehosephat=0
X=987654321
```

Dynamic Formats (3)

I referred to ignoring spaces being very useful
Let's see the format it creates:

```
CALL trivial ('fred', 12345)

' (A, "=", I                    5)'
```

Even more useful when varying m and k in
Fn.m, ESn.m, ESn.mEk etc.

Free-format Input (1)

You can actually do quite a lot in Fortran
But it often needs some very nasty tricks

- You can read arbitrarily long lines as text

And then decode them using character operations
See the extra, extra I/O lecture for an incantation

Think about whether it is the best approach
There are several, possibly simpler, alternatives

Free-format Input (2)

- Use a separate Python program to read it

Write it out in a Fortran–friendly fixed–format form

Probably the easiest for ‘true’ free–format
There are courses on this, and I do it
You can wrap it in a simple shell script
The Python program can also call the Fortran one

See
“Building Applications out of Multiple Programs”

Free-format Input (3)

You could also use Perl or anything else

Calling Python is possible, but fairly hairy
Generally, I don't recommend doing it

- Call a C function to read it

It's easy only for people who know C well

Calling C is covered in an extra lecture
It's not hard, but there are a lot of "gotchas"

Free-format In Fortran

Now we get back to using only Fortran

- Firstly, is the layout under your control?

Either, can you edit the program that writes it?
Or, is it being input by a human?

Let's assume that the answers are "yes"
The following is what can be done very simply

You Control Both Codes

- Use only list–directed input formats
- Ensure that all items are of the same type or a uniform repetition (see example 2)
- Don't end the items part–way through a line

And any one of:

- There are a a known number of items
- Each line has a known number of items and the termination is by end–of–file
- You terminate each list with a '/'

Example (1)

```
REAL :: X(10)
READ *, N, (X(I), I = 1,N)
PRINT *, (X(I), I = 1,N)
```

```
3    1.23    4.56    7.89
```

Produces a result like:

```
1.2300000   4.5600000    7.8900000
```

Example (2)

```
CHARACTER(LEN=8) :: Z(10)
REAL :: X(10)
READ *, N, (Z(I), X(I), I = 1,N)
PRINT *, (Z(I), X(I), I = 1,N)
```

```
3 Fred 1.23 Joe 4.56
        Bert 7.89
```

Produces a result like:

```
Fred 1.2300000 Joe 4.5599999 Bert 7.8899999
```

Example (3)

```
REAL :: X(10)
X = -1.0
DO I = 1, 10, 3
    READ (*, *, END=99) X(I:MIN(I+2,10))
END DO
99 PRINT *, X

1.23    2.34    3.45
4.56  5.67  6.78
```

Produces a result like:

```
1.23 2.34 3.45 4.56 5.67 6.78 -1.0 -1.0 -1.0 -1.0
```

Example (4)

```
REAL :: X(10)
X = -1.0
READ (*, *) X
PRINT *, X

1.23        4.56
   7.89        0.12    /
```

Produces a result like:

```
1.23 4.56 7.89 0.12 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0
```

CSV (1)

Comma Separated Values – e.g. RFC 4180
http://en.wikipedia.org/wiki/Comma-separated_values/

Reading CSV can be from easy to foul

Simple way is to read whole record as text
Concatenate a slash ('/') and use list-directed

```
CHARACTER(LEN=1000) :: buffer
READ (5, '(A)') buffer
READ ( buffer+"/" , *) <variables>
```

CSV (2)

Main problem is unquoted text containing any of:
asterisk, slash, apostrophe, quote or space

Can sometimes be read but may cause chaos
Fortran's rules and CSV's are bizarre and different

Using Python to sanitise it is the best method
Check it carefully for sanity when you do that

CSV (3)

Writing is usually easy, if somewhat tedious
IF the reading program ignores layout spaces!

Preventing unwanted newlines needs a bit of care
E.g. '(1000000(A0,",",I0,",",5(",",ES0.9),:))'
Note the use of the colon to avoid a trailing comma

A fairly good practical exercise in formatted I/O
Remember to experiment with quoting strings

Alternative Exception Handling

You can use END=<label> or ERR=<label>
Does a GOTO <label> on the relevant event

IOSTAT is generally cleaner and more ‘modern’

Fortran 2003 IOMSG returns a text message

- It does not of itself trap errors or EOF

```
CHARACTER(LEN=120) :: iomsg
OPEN (1, FILE='fred', IOSTAT=ioerr, IOMSG=iomsg)
IF (IOSTAT /= 0) PRINT *, iomsg
```

OPEN Specifier RECL

This specifies the file's record length
It is mandatory for direct–access I/O
You rarely need to set it for sequential I/O

The default for unformatted is usually $2^{31}-1$
Maximum under all systems you will meet

The formatted default is from 132 upwards
You may need to increase it if it is too small
Don't go overboard, as it allocates a buffer

Other OPEN Specifiers

DELIM – see under list–directed I/O

POSITION can be ‘asis’, ‘rewind’ or ‘append’
Sets initial position in file – you rarely need to

STATUS has its uses, but you can ignore it
Except for scratch files, as described
It doesn’t do what most people think that it does
But, in Fortran 77, it was all that there was
Recommended to use ACTION as a better alternative

There are others, but they are rarely useful

Updating Existing Files

When a WRITE statement is executed:

- Sequential files are always truncated
Immediately following the record just written

- Direct–access files are never truncated
The record is replaced in place

End of (Fortran 90) story
Fortran 2003 allows some control over it

REWIND (1)

This is available for sequential I/O only

Almost nobody has major problems

Repositions back to the start of the file

- Allows changing between READ and WRITE

Commonly used for workspace ('scratch') files

- Don't rewind files opened for APPEND

Applies to all languages on modern systems

REWIND (2)

```
DO . . . write out the data . . .
    WRITE (17) . . .
END DO
REWIND (17)
DO . . . read it back again . . .
    READ (17) . . .
END DO
REWIND (17)
DO . . . and once more . . .
    READ (17) . . .
END DO
```

Direct-Access I/O is Simple

Very few users have any trouble with it

- It is simpler and cleaner than C's

Most problems come from "thinking in C"
But some come from "being too clever by half"

- Use only unformatted direct–access I/O

Formatted works, but is trickier and rarely used

Direct-Access (1)

The model is that of fixed–length records

OPEN sets the length in (effectively) bytes

- You must set the length in the OPEN
- You must reopen files with the same length
- INQUIRE can query it only after OPEN

This is needed because of the I/O model conflict

Direct-Access (2)

Each record is referred to by its number

Records are created simply by being written
Files will be extended automatically, if needed

- Don't read a record until it has been written
- Don't use sparse record numbers

Implementing sparse indexing isn't hard
But ask for help if you need to do it

Example (1)

```
REAL, DIMENSION(4096) :: array = 0.0

OPEN (1, FILE='fred', ACCESS='direct', &
        ACTION='write', FORM='unformatted',
        RECL=4*4096)

DO k = 1,100
        WRITE (1, REC=k) array
END DO
. . .
```

That is the best way to initialise such a file

Example (2)

Opening a read–only direct–access file

```
REAL, DIMENSION(4096) :: array = 0.0

OPEN (1, FILE='fred', ACCESS='direct', &
      ACTION='read', FORM='unformatted', &
      RECL=4*4096)
...
READ (1, REC=<expr>) array
...
```

Example (3)

Opening a direct–access file for update

```
OPEN (1, FILE='fred', ACCESS='direct', &
      FORM='unformatted', RECL=4*4096)
. . .
READ (1, REC=k) array
WRITE (1, REC=INT(array(1))) array
READ (1, REC=INT(array(2))) array
. . .
```

Note the mixing of READ and WRITE

Programming Notes

- Each transfer may cause a system call
And potentially an actual disk access

- Use large records, as for unformatted I/O

Unix has a system file cache for open files
No major efficiency problems while files fit
Can be major performance problems when not

- Ask for help if you hit trouble here

And There's More . . .

There are some slides on yet more facilities
More to tell you what exists than teach them

Non–advancing I/O is very useful for free–format

INQUIRE queries properties of files, units etc.

And so on . . .

Features Not Covered

There are extra slides on:

- Data pointers (not much used in Fortran)
- Arrays, procedures and yet more I/O

Completely omitted topics in Fortran 95 TRs:

- Varying strings
- Preprocessing
- IEEE 754 exception handling (in Fortran 2003)

- Lots of more obscure features and details
- Anything that I recommend not using

Fortran 2003

- Dozens of Fortran 95 restrictions removed
- Full object orientation
- Some semantic extension features
- Parameterised derived types
- Procedure pointers
- ASSOCIATE (a sort of cleaner macro)
- System interfaces (e.g. command args)
- Interfacing with C etc.
- And yet more ...