Introduction to Modern Fortran

I/O and Files

Nick Maclaren

Computing Service

nmm1@cam.ac.uk, ext. 34761

November 2007

I/O Generally

Most descriptions of I/O are only half–truths
Those work most of the time – until they blow up
Most modern language standards are like that

Fortran is rather better, but there are downsides
Complexity and restrictions being two of them

- Fortran is much easier to use than it seems
- This is about what you can rely on in practice

We will start with the basic principles

Some ‘Recent’ History

Fortran I/O (1950s) predates even mainframes
OPEN and filenames was a CC† of c. 1975

Unix/C spread through CS depts 1975–1985
ISO C’s I/O model was a CC† of 1985–1988
Modern languages use the C/POSIX I/O model
Even Microsoft systems are like Unix here

- The I/O models have little in common

† CC = committee compromise

Important Warning

It is often better than C/C++ and often worse
But it is very different at all levels

- It is critical not to think in C–like terms

Trivial C/C++ tasks may be infeasible in Fortran

As always, use the simplest code that works
Few people have much trouble if they do that

- Ask for help with any problems here

Fortran's Sequential I/O Model

A Unix file is a sequence of characters (bytes)
A Fortran file is a sequence of records (lines)

For simple, text use, these are almost equivalent

In both Fortran and C/Unix:

- Keep text records short (say, < 250 chars)
- Use only printing characters and space
- Terminate all lines with a plain newline
- Trailing spaces can appear and disappear

What We Have Used So Far

To remind you what you have been doing so far:

PRINT *, can be written WRITE (*,*)

READ *, can be written READ (*,*)

READ/WRITE (*,*) is shorthand for

READ/WRITE (UNIT=*, FMT=*)

READ *, ... and PRINT *, ... are legacies

Their syntax is historical and exceptional

Record-based I/O

- Each READ and WRITE uses 1+ records

Any unread characters are skipped for READ
WRITE ends by writing an end–of–line indicator

- Think in terms of units of whole lines

A WRITE builds one or more whole lines
A READ consumes one or more whole lines

Fortran 2003 relaxes this, to some extent

Fortran's I/O Primitives

All Fortran I/O is done with special statements
Any I/O procedure is a compiler extension

Except as above, all of these have the syntax:

<statement> (<control list>) <transfer list>

The <transfer list> is only for READ and WRITE

The <control list> items have the syntax:

<specifier>=<value>

Translation

An I/O statement is rather like a command

A <control list> is rather like a set of options
Though not all of the specifiers are optional

The <transfer list> is a list of variables to read
or a list of expressions to write

We now need to describe them in more detail

Specifier Values (1)

All specifier values can be expressions
If they return values, they must be variables

- Except for * in UNIT=* or FMT=*

Even lunatic code like this is permitted

```
INTEGER, DIMENSION(20) :: N
CHARACTER(LEN=50) :: C

WRITE (UNIT = (123*K)/56+2, FMT = C(3:7)//')', &
      IOSTAT=N(J**5-15))
```

Specifier Values (2)

The examples will usually use explicit constants

```
OPEN (23, FILE='trace.out', RECL=250)
```

- But you are advised to parameterise units

And anything else that is system dependent
Or you might need to change later

```
INTEGER, PARAMETER :: tracing = 23, tracelen = 250
CHARACTER(LEN=*), PARAMETER :: &
    tracefile = 'trace.out'

OPEN (tracing, FILE=tracefile, RECL=tracelen)
```

Basics of READ and WRITE

READ/WRITE (<control list>) <transfer list>

Control items have form <specifier> = <value>
UNIT is the only compulsory control item
The UNIT= can be omitted if the unit comes first

The unit is an integer identifying the connection
It can be an expression, and its value is used

UNIT=* is an exceptional syntax
It usually means stdin and stdout

Transfer Lists

A list is a comma–separated sequence of items
The list may be empty (READ and WRITE)

- A basic output item is an expression
- A basic input item is a variable

Arrays and array expressions are allowed

- They are expanded in array element order

Fancy expressions will often cause a copy
Array sections should not cause a copy

Example

```
REAL :: X(10)
READ *, X(:7)
PRINT *, X(4:9:3)*1000.0

1.23 2.34 3.45 4.56 5.67 6.78 7.89
```

Produces a result like:

```
4.5600000E+03   7.8900000E+03
```

Empty Transfer Lists

These are allowed, defined and meaningful

READ (*, *) skips the next line

WRITE (*, *) prints a blank line

WRITE (*, FORMAT) prints any text in FORMAT

That may print several lines

A Useful Trick

A useful and fairly common construction

```
INTEGER :: NA
REAL, DIMENSION(1:100) :: A
READ *, NA, A(1:NA)
```

Fortran evaluates a transfer list as it executes it

- Be warned: easy to exceed array bounds

At least, you should check the length afterwards
Safer to put on separate lines and check first

Implied DO-loops

There is an alternative to array expressions
Equivalent, but older and often more convenient

Items may be (<list> , <indexed loop control>)
This repeats in the loop order (just like DO)

((A(I,J) , J = 1,3) , B(I), I = 6,2,–2)

A(6,1), A(6,2), A(6,3), B(6), A(4,1), A(4,2),
A(4,3), B(4), A(2,1), A(2,2), A(2,3), B(2)

Programming Notes

You can do I/O of arrays in three ways:

- You can write a DO–loop around the I/O
- Array expressions for selecting and ordering
- You can use implied DO–loops

Use whichever is most convenient and clearest
There are no problems with combining them
More examples of their use will be shown later

There isn't a general ranking of efficiency

The UNIT Specifier

- A unit is an integer value

Except for UNIT=*, described above

It identifies the connection to a file

- UNIT= can be omitted if the unit is first

A unit must be connected to a file before use

Generally use values in the range 10–99

- That's all you need to know for now

The FMT Specifier

This sets the type of I/O and must match the file

- FMT= can be omitted if the format is second and the first item is the unit

- FMT=* indicates list–directed I/O
- FMT=<format> indicates formatted I/O

These can be interleaved on formatted files

- No FMT specifier indicates unformatted I/O

Example

These are formatted I/O statements

```
WRITE (UNIT = *, FMT = '(2F5.2)') c
READ (99, '(F5.0)') x
WRITE (*, FMT = myformat) p, q, r
```

These are list–directed I/O statements

```
WRITE (UNIT = *, FMT = *) c
READ (99, *) x
```

These are unformatted I/O statements

```
WRITE (UNIT = 64) c
READ (99) x
```

List-Directed Output (1)

What you have been doing with 'PRINT *,'

The transfer list is split into basic elements
Each element is then formatted appropriately
It is separated by spaces and/or a comma

- Except for adjacent CHARACTER items

Write spaces explicitly if you want them

The format and layout are compiler–dependent

Example

```
REAL :: z(3) = (/4.56, 4.56, 4.56/)
CHARACTER(LEN=1) :: c = 'a'
PRINT *, 1.23, 'Oh dear', z, c, '"', c, ' ', c, c
```

Produces (under one compiler):

```
1.2300000 Oh dear 4.5599999 4.5599999
4.5599999 a"a aa
```

List-Directed Output (2)

You can cause character strings to be quoted
Very useful if writing data for reinput
Use the DELIM specifier in the OPEN

```
OPEN (11, FILE='fred', DELIM='quote')
WRITE (11, *) 'Kilroy was here'
```

```
"Kilroy was here"
```

Also DELIM='apostrophe' and DELIM='none'
Fortran 2003 allows them in the WRITE, too

List-Directed Input (1)

What you have been doing with ‘READ *,’

This does the reverse of ‘PRINT *,’
The closest Fortran comes to free–format input

- It automatically checks the data type

- OK for lists of numbers and similar
Not much good for genuinely free–format

List-Directed Input (2)

Strings may be quoted, or not
Using either quote (") or apostrophe (')

- Quote all strings containing the following:
  , / " ' * space end–of–line

For the reasons why, read the specification
List–directed input is actually quite powerful
But very unlike all other modern languages

Example

```
REAL :: a, b, c
CHARACTER(LEN=8) :: p, q
READ *, a, p, b, q, c
PRINT *, a, p, b, q, c
```

```
123e–2 abcdefghijkl –003 "P""Q'R" 4.56
```

Produces (under one compiler):

```
1.2300000 abcdefgh –3.0000000 P"Q'R
4.5599999
```

Free-Format

Free–format I/O is not traditional in Fortran

Formatted output is far more flexible
Fortran 2003 adds some free–format support

Free–format input can be very tricky in Fortran
But it isn't hard to read lists of numbers

There is some more on this in extra slides

Unformatted I/O is Simple

Very few users have any trouble with it

- It is NOT like C binary I/O
- It is unlike anything in C

Most problems come from "thinking in C"

Unformatted I/O (1)

- It is what you use for saving data in files

E.g. writing your own checkpoint/restart
Or transferring bulk data between programs

- No formatting/decoding makes it a lot faster

100+ times less CPU time has been observed

- Assume same hardware and same system

If not, see other courses and ask for help

Unformatted I/O (2)

Just reads and writes data as stored in memory

- You must read back into the same types

- Each transfer uses exactly one record

With extra control data for record boundaries
You don't need to know what it looks like

- Specify FORM='unformatted' in OPEN

stdin, stdout and terminals are not suitable

That's ALL that you absolutely need to know!

Example

```
INTEGER, DIMENSION(1000) :: index
REAL, DIMENSION(1000000) :: array

OPEN (31, FILE='fred', FORM='unformatted')

DO k = 1,...
    WRITE (31) k, m, n, index(:m), array(:n)
END DO
```

In another run of the program, or after rewinding:

```
DO k = 1,...
    READ (31) junk, m, n, index(:m), array(:n)
END DO
```

Programming Notes

- Make each record (i.e. transfer) quite large

But don’t go over 2 GB per record

- I/O with whole arrays is generally fastest

```
INTEGER :: N(1000000)
READ (29) N
```

Array sections should be comparably fast

- Remember about checking for copying

- Implied DO–loops should be avoided

At least for large loop counts

Formatted I/O

READ or WRITE with an explicit format
A format is just a character string
It can be specified in any one of three ways:

- A CHARACTER expression
- A CHARACTER array
  Concatenated in array element order
- The label of a FORMAT statement
  Old–fashioned, and best avoided

Formats (1)

A format is items inside parentheses

Blanks are ignored, except in strings

‘ (i3,f   5   .   2)   ’ ≡ ‘(i3,f5.2)’

We will see why this is so useful later

Almost any item may have a repeat count

‘(3 i3, 2 f5.2)’ ≡ ‘(i3, i3, i3, f5.2, f5.2)’

Formats (2)

A group of items is itself an item
Groups are enclosed in parentheses

E.g. '(3 (2 i3, f5.2))' expands into:
'(i3, i3, f5.2, i3, i3, f5.2, i3, i3, f5.2)'

Often used with arrays and implied DO–loops

Nesting them deeply can be confusing

Example

```
REAL, DIMENSION(2, 3) :: coords
INTEGER, DIMENSION(3) :: index

WRITE (29, '( 3 ( i3, 2 f5.2 ) )}') &
      ( index(i), coords(:, i), i = 1,3)
```

This is how to use a CHARACTER constant:

```
CHARACTER(LEN=*), PARAMETER :: &
      format = '( 3 ( i3, 2 f5.2 ) )'

WRITE (29, format) ( index(i), coords(:, i), i = 1,3)
```

Transfer Lists And Formats

Logically, both are expanded into flat lists
I.e. sequences of basic items and descriptors

The transfer list is the primary one
Basic items are taken from it one by one
Each then matches the next edit descriptor

The item and descriptor must be compatible
E.g. REAL vars must match REAL descs

Input Versus Output

We shall mainly describe formatted output
This is rather simpler and more general

Unless mentioned, all descriptions apply to input
It's actually much easier to use than output
But it is rather oriented to form–filling

More on flexible and free–format input later

Integer Descriptors

In (i.e. letter i) displays in decimal
Right–justified in a field of width n
In.m displays at least m digits

WRITE (*, '(I7)') 123 ⇒ '   123'
WRITE (*, '(I7.5)') 123 ⇒ ' 00123'

You can replace the I by B, O and Z
For binary, octal and hexadecimal

Example

```
WRITE (*, '( I7, I7 )') 123, -123
WRITE (*, '( I7.5, I7.5 )') 123, -123

    123   -123
  00123 -00123

WRITE (*, '( B10, B15.10 )') 123, 123
WRITE (*, '( O7, O7.5 )') 123, 123
WRITE (*, '( Z7, Z7.5 )') 123, 123

   1111011     0001111011
    173  00173
     7B  0007B
```

Values Too Large

This is field overflow on output
The whole field is replaced by asterisks

Putting 1234 into i4 gives 1234
Putting 12345 into i4 gives ****
Putting –123 into i4 gives –123
Putting –1234 into i4 gives ****

This applies to all numeric descriptors
Both REAL and INTEGER

Fixed-Format REAL

Fn.m displays to m decimal places
Right–justified in a field of width n

```
WRITE (*, '( F9.3 )') 1.23        ⇒ '    1.230'
WRITE (*, '( F9.5 )') 0.123e-4    ⇒ '  0.00001'
```

You may assume correct rounding
Not required, but traditional in Fortran

- Compilers may round exact halves differently

Widths of Zero

For output a width of zero may be used
But only for formats I, B, O, Z and F
It prints the value without any leading spaces

```
write (*, '("/",i0,"/",f0.3)') 12345, 987.654321
```

Prints

```
/12345/987.65
```

Exponential Format (1)

There are four descriptors: E, ES, EN and D
With the forms En.m, ESn.m, ENn.m and Dn.m

All of them use m digits after the decimal point
Right–justified in a field of width n

D is historical – you should avoid it
Largely equivalent to E, but displays D

For now, just use ESn.m – more on this later

Exponential Format (2)

The details are complicated and messy
You don't usually need to know them in detail
Here are the two basic rules for safety

- In En.m and ESn.m, make n $\geq$ m+7

That's a good rule for other languages, too

- Very large or small exponents display oddly

I.e. exponents outside the range –99 to +99
Reread using Fortran formatted input only

Numeric Input

F, E, ES, EN and D are similar

- You should use only Fn.0 (e.g. F8.0)

For extremely complicated reasons

- Any reasonable format of value is accepted

There are more details given later

CHARACTER Descriptor

An displays in a field with width n
Plain A uses the width of the CHARACTER item

On output, if the field is too small:
    The leftmost characters are used
Otherwise:
    The text is right–justified

On input, if the variable is too small:
    The rightmost characters are used
Otherwise:
    The text is left–justified

Output Example

```
WRITE (*,'(a3)') 'a'
WRITE (*,'(a3)') 'abcdefgh'
```

Will display:

```
  a
abc
```

Input Example

```
CHARACTER(LEN=3) :: a

READ (*,'(a8)') a ;    WRITE (*,'(a)') a
READ (*,'(a1)') a ;    WRITE (*,'(a)') a
```

With input:

```
abcdefgh
a
```

Will display:

```
fgh
a
```

LOGICAL Descriptor

Ln displays either T or F
Right–justified in a field of width n

On input, the following is done

- Any leading spaces are ignored
- An optional decimal point is ignored
- The next char. must be T (or t) or F (or f)
- Any remaining characters are ignored

E.g. '.true.' and '.false.' are acceptable

The G Descriptor

The G stands for generalized
It has the forms Gn or Gn.m
It behaves according to the item type

INTEGER behaves like In
CHARACTER behaves like An
LOGICAL behaves like Ln
REAL behaves like Fn.m or En.m
depending on the size of the value

The rules for REAL are fairly sensible

Other Types of Descriptor

All of the above are data edit descriptors
Each of them matches an item in the transfer list
As mentioned, they must match its type

There are some other types of descriptor
These do not match a transfer list item
They are executed, and the next item is matched

Text Literal Descriptor

A string literal stands for itself, as text
It is displayed just as it is, for output
It is not allowed in a FORMAT for input

Using both quotes and apostrophes helps
The following are all equivalent

```
WRITE (29, '( "Hello" )')
WRITE (29, "( 'Hello' )")
WRITE (29, '( ''Hello'' )')
WRITE (29, "( ""Hello"" )")
```

Spacing Descriptor

X displays a single blank (i.e. a space)
It has no width, but may be repeated

On input, it skips over exactly one character

```
READ (*, '(i1, 3x, i1)') m, n
WRITE (*, '(i1, x, i1, 4x, a)') m, n, '!'

7PQR9
```

Produces '7 9   !'

Newline Descriptor (1)

/ displays a single newline (in effect)
It has no width, but may be repeated

It can be used as a separator (like a comma)
Only if it has no repeat count, of course

```
WRITE (*, '(i1/i1, 2/, a)') 7, 9, '!'

7
9

!
```

Newline Descriptor (2)

On input, it skips the rest of the current line

```
READ (*, '(i1/i1, 2/, i1)') l, m, n
WRITE (*, '(i1, 1x, i1, 1x, i1)') l, m, n

1 1 1 1
2 2 2 2
3 3 3 3
4 4 4 4
```

Produces “1 2 4”

Item-Free FORMATs

You can print multi–line text on its own

```
WRITE (*, '("Hello" / "Goodbye")')

Hello
Goodbye
```

And skip as many lines as you like

```
READ (*, '(////)')
```

Generalising That

That is a special case of a general rule

FORMATs are interpreted as far as possible

```
WRITE (*, '(I5, " cubits", F5.2)') 123

  123 cubits
```

This reads 42 and skips the following three lines

```
READ (*, '(I3///)') n

42
```

Complex Numbers

For list–directed I/O, these are basic types
E.g. read and displayed like “(1.23,4.56)”

For formatted and unformatted I/O
COMPLEX numbers are treated as two REALs
Like an extra dimension of extent two

```
COMPLEX :: c = (1.23, 4.56)
WRITE (*, '(2F5.2,3X,2F5.2)') c, 2.0*c
```

```
 1.23  4.56     2.46  9.12
```

Exceptions and IOSTAT (1)

By default, I/O exceptions halt the program
These include an unexpected end–of–file

You trap by providing the IOSTAT specifier

```
INTEGER :: ioerr

OPEN (1, FILE='fred', IOSTAT=ioerr)

WRITE (1, IOSTAT=ioerr) array

CLOSE (1, IOSTAT=ioerr)
```

Error Handling and IOSTAT (2)

IOSTAT specifies an integer variable

Zero means success, or no detected error

Positive means some sort of I/O error
An implementation should describe the codes

Negative means end–of–file (but see later)
In Fortran 2003, this value is IOSTAT_EOF

What Is Trapped? (1)

The following are NOT errors
Fortran defines all of this behaviour

- Formatted READ beyond end–of–record

Padded with spaces to match the format

Fortran 2003 allows a little control of that

- Writing a value too large for a numeric field

The whole field is filled with asterisks (*****)

What Is Trapped? (2)

The following are NOT errors

- Writing too long a CHARACTER string

The leftmost characters are used

- Reading too much CHARACTER data

The rightmost characters are used

What Is Trapped? (3)

The following is what you can usually rely on

- End–of–file

- Unformatted READ beyond end–of–record
IOSTAT may be positive OR negative

- Most format errors (syntactically bad values)
E.g. 12t8 being read as an integer

That is roughly the same as C and C++

What Is Trapped? (4)

The following are sometimes trapped
The same applies to most other languages

- Numeric overflow (integer or floating–point)

Floating–point overflow may just deliver infinity
Integer overflow may wrap modulo 2^N
Or there may be even less helpful effects

- 'Real' (hardware or system) I/O errors

E.g. no space on writing, file server crashing
Anything may happen, and chaos is normal

2 GB Warning

I said ‘‘chaos is normal’’ and meant it
Be careful when using files of more than 2 GB

Most filesystems nowadays will support such files
But not all of the interfaces to them do
Things like pipes and sockets are different again

That has nothing to do with Fortran, as such

Different compilers may use different interfaces
And there may be options you have to specify

OPEN

Files are connected to units using OPEN

```
OPEN (UNIT=11, FILE='fred', IOSTAT=ioerr)
```

That will open a sequential, formatted file
You can then use it for either input or output

You can do better, using optional specifiers
Other types of file always need one or more

Choice of Unit Number

Unit numbers are non–negative integer values
The valid range is system–dependent
You can usually assume that 1–99 are safe

Some may be in use (e.g. for stdin and stdout)
They are often (not always) 5 and 6

It is best to use unit numbers 10–99
Most codes just do that, and have little trouble

ACCESS and FORM Specifiers

These specify the type of I/O and file

'sequential' (default) or 'direct'
'formatted' (default) or 'unformatted'

```
OPEN (UNIT=11, FILE='fred', ACCESS='direct', &
    FORM='unformatted', RECL=500, IOSTAT=ioerr)
```

That will open a direct–access, unformatted file
with a record length of 500
You can then use it for either input or output

Scratch Files

```
OPEN (UNIT=11, STATUS='scratch', &
    FORM='unformatted', IOSTAT=ioerr)
```

That will open a scratch (temporary) file
It will be deleted when it is closed

It will be sequential and unformatted
That is the most common type of scratch file
But all other types and specifiers are allowed

- Except for the FILE specifier

The ACTION Specifier

- This isn’t needed, but is strongly advised

It helps to protect against mistakes

It enables the reading of read–only files

```
OPEN (UNIT=11, FILE='fred', ACTION='read', &
    IOSTAT=ioerr)
```

Also ‘write’, useful for pure output files

The default, ‘readwrite’, allows both

Example (1)

Opening a text file for reading data from

```
OPEN (UNIT=11, FILE='fred', ACTION='read', &
      IOSTAT=ioerr)
```

Opening a text file for writing data or results to

```
OPEN (UNIT=22, FILE='fred', ACTION='write', &
      IOSTAT=ioerr)
```

```
OPEN (UNIT=33, FILE='fred', ACTION='write', &
      RECL=80, DELIM='quote', IOSTAT=ioerr)
```

Example (2)

Opening an unformatted file for reading from

```
OPEN (UNIT=11, FILE='fred', ACTION='read', &
      FORM='unformatted', IOSTAT=ioerr)
```

Opening an unformatted file for writing to

```
OPEN (UNIT=22, FILE='fred', ACTION='write', &
      FORM='unformatted', IOSTAT=ioerr)
```

Example (3)

Opening an unformatted workspace file
It is your choice whether it is temporary

```
OPEN (UNIT=22, STATUS='scratch', &
    FORM='unformatted', IOSTAT=ioerr)

OPEN (UNIT=11, FILE='/tmp/fred', &
    FORM='unformatted', IOSTAT=ioerr)
```

See extra slides for direct–access examples

Omitted For Sanity

These are in the extra slides

Techniques for reading free–format data
Some more detail on formatted I/O
Internal files and dynamic formats
More on OPEN, CLOSE, positioning etc.
Direct–access I/O

There are extra, extra slides on some details