Fortran 90 Input and Output (I/O)

- The input/output system translates data between user-readable character form and internal binary form in the computer memory.
- List-directed (or Default) format

  READ *, *variable list*
  PRINT *, *expression list*

  } list items separated by commas

  – This is the method we have used so far.
  – I/O uses default devices (keyboard and screen)
  – The * means use list-directed I/O, I.e. read or write according to the data types of the variables in the list for input, and the data types of the expressions (including variables and constants) on output.

List-directed I/O (*cont.*)

- Input data items are constants agreeing in type with variables in the list. (May read integer into real variable but not *v.v.*)
  - Character data must be in single or double quotes unless it is essentially a single 'word'.
  - Items are separated by blank, comma, or RETURN.
    - (see textbook, *e.g.* Ellis p238, for the full details and conditions for input)
- Output starts at the beginning of a line.
  - The output format for numerical data and the spacing between items depends on the compiler.
  - All significant figures of the data types are output.
  - Character constants are normally output without enclosing quotes.
  - To output a blank line use **`PRINT *`** (no comma)

List-directed I/O (*cont.*)

- List-directed *input*
  - Most suitable for entering numerical data from the keyboard.
  - No rigidity in spacing of data items.
  - Character data may have to be in quotes which is a nuisance.
  - It may be less suitable for reading tabulated data from files when the format is fixed.
- List-directed *output*
  - Useful for initial testing and for messages (character constants).
  - But programmer has no control over form of numerical output, *e.g.* number of significant figures, and layout.
- Explicit Input/Output formatting solves these problems.

Formatted Output

PRINT '(*format list***)', *expression list***

- where **(***format list***)** is a character constant describing how the output list is to be displayed.
- The format list consists of a series of editing descriptors separated by commas and in parentheses.

e.g. **PRINT '(1X,A,I2,F7.3)', "Answer", N, RESULT**

- It agrees in order and type with the output list, but may contain additional descriptors for spacing.
- Alternatively (useful if you need the same format list for more than one **PRINT** statement, or want to modify the format at run-time)

PRINT *fmt*, *expression list*

fmt is a character variable or named constant for the format list.

e.g.
```
CHARACTER(LEN=14)[, PARAMETER] :: &
          FORMAT_1 = "(1X,A,I2,F7.3)"
   PRINT FORMAT_1, "Answer",  N,  RESULT
```

Formatted Output (*cont.*)

- WARNING
- The standard specifies that the first character of the output line should be a "carriage control" character. This is used to control printed output and is a historical leftover.
  - This first character may or may not appear on the screen. Compilers vary on how they handle this control code.
  - List-directed output automatically inserts a space at the beginning of the line.

FOR FORMATTED OUTPUT THE FIRST CHARACTER SHOULD BE A SPACE

Format types

- A format -- character editing

  A*w* **A** alphameric

  w field width

  If *w* > *len* (the number of characters in the output list item)
  leading spaces are inserted (right-justified).

  If *w* < *len* only the leftmost *w* characters are output.

  If *w* is omitted exactly *len* characters are output.

  len is the number of characters in a constant `'ABCDE'`

  the number of characters in a substring `VAR(7:10)`

  or the declared length of a character variable `VAR`

```
CHARACTER(LEN=12) :: VAR
```

Format types (*cont.*)

- Examples of A format

```
PRINT *, "Simple text"      ⇒ Simple text
PRINT "(A)", "Simple text" ⇒ imple text*
CHARACTER(LEN=5) :: FMT = "(A15)"
PRINT FMT, "Simple text"    ⇒ Simple text
FMT = "(A10)"
PRINT FMT, "Simple text"    ⇒ imple tex*
CHARACTER(LEN=20) :: VAR = "Simple text"
PRINT '(A)', VAR            ⇒ imple text* + 9 spaces
PRINT '(A)', VAR(8:11)      ⇒ text
```

*Note first character may be missing , but OK in Salford output.

Add a space at the beginning of the character string -- or use

- X format -- blank editing

*n***X** inserts *n* spaces in output line

```
PRINT "(1X,A)", "Simple text"
```

Format types (*cont.*)

- I format -- integer editing
  for INTEGER expression

`I`*w* *w* field width in characters. Must be given.

Must include all possible digits
including sign if negative. *e.g.* `I4` `-123`
Output right justified in field. `!!!3`

Number too large for format. *e.g.* `12345` `****`

`I`*w*.*m* Will output minimum *m* digits *e.g.* `I4.2` `!!03`

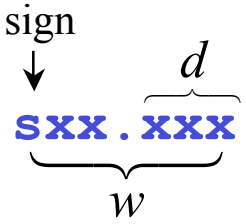
`!` represents a space in output.

Format types (*cont.*)

- F format -- for REAL/DOUBLE PRECISION decimal form (no exponent)

  F*w.d* *w* includes decimal point and sign
  d is number of decimal places

  Number is rounded to *d* places



$w \geq d + 3$

If < *d* decimal places in number zeros are appended.

Output right-justified in field.

e.g `F6.2` `-12.34` `123.45` `_1.23`

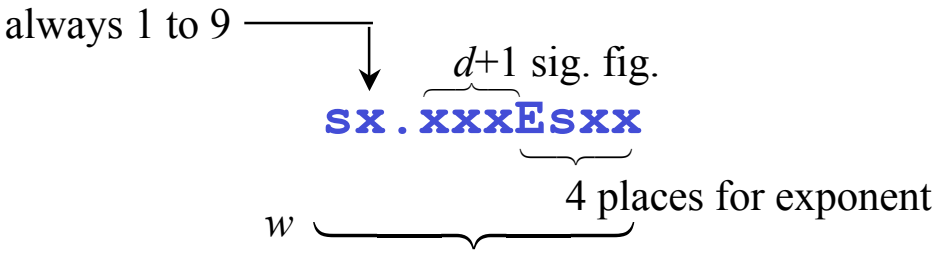
> *w* characters `1234.56` ⇒ `******`

Format types (*cont.*)

- ES format -- for REAL/DOUBLE PRECISION scientific form

F format not suitable for very large or very small numbers.
Use scientific notation, mantissa normalized to range 1 to 10.

ES*w.d* *w* includes decimal point, exponent (power of 10) and signs for both value and exponent.



e.g 25.3	`ES10.2`	`¦¦2.53E+01`	`F6.2`	`¦25.30`
-0.314159	`ES11.4`	`-3.1416E-01`	`F8.3`	`¦¦-0.314`
-12.6×10^{83}	`ES11.3`	`¦-1.260E+84`	`F11.3`	`****...*`

For exponents > 99 see textbooks.

Format types (*cont.*)

- E format -- for REAL/DOUBLE PRECISION traditional exponent form
  - Same as ES, except
    Mantissa in range 0.1 to 1
    Digit before decimal point always 0
    (or absent in some implementations)
- Formatting example

```
PRINT "(1X,   A,   I3,  A, F6.3, A, ES10.3)", &
   "The result of", K, " x", PI, " is", R
```

Given `INTEGER K   REAL PI, R   K=25  PI=3.142`

Output is

```
 The result of 25 x 3.142 is 7.855E+01
```

Format types (*cont.*)

- Repetitions
  - An **`A`**, **`I`**, **`F`**, **`ES`**, **`E`** format descriptor may be preceded by a number showing how many times it is to be used.
    e.g. **`(I5, I5, I5)`** ⇒ **`(3I5)`**
  - Format groups may also be repeated, using parentheses
    e.g. **`(1X, A, F6.3, A, F6.3)`** ⇒ **`(1X, 2(A,F6.3))`**

MAKE SURE YOU HAVE A FORMAT DESCRIPTOR FOR EVERY ITEM IN THE ITEM LIST OF YOUR PRINT OR WRITE STATEMENT.

MAKE SURE THAT THE FORMAT DESCRIPTORS MATCH, IN ORDER, THE DATA TYPES OF THE ITEMS IN THE LIST

  - See textbooks for further details and other format types.

Formatted input

The general form of a READ statement using the keyboard is

READ *fmt, variable list*

where *fmt* is an ***** implying default or list-directed input
or the *format list* as a character constant or variable.

For keyboard input of numerical data list-directed input is preferred.

But, to read character data only, without restrictions and enclosing it in quotes, use **A** format

e.g.
```
CHARACTER(LEN=20) :: STRING
READ '(A)', STRING
```

will read up to 20 characters from the keyboard into **STRING**.
Any more will be ignored.

This enables a whole line of input from the keyboard, including any characters, to be read into a string variable. If necessary the variable will be padded with blanks.

Formatted input (*cont.*)

- Formatted input may be used from files where data is laid out in a tabular form.
- **A***w* → *w* characters into character variable, length *len*
  If *w* < *len*, variable padded with blanks at end,
  If *w* > *len*, rightmost *len* characters of input string stored in variable.
- *n***X** → Ignore next *n* characters.
- **I***w* → Right-justified integer into INTEGER variable.
- **F***w.d* → Next *w* characters contain a number in decimal or **E***w.d* exponential form to be read into REAL or DOUBLE PRECISION variable.
  A decimal point in the input data overrides the *d*.
  Always include one.

Formatted input (*cont.*)

- Example of input:

```
 The result of 25 x 3.142 is 7.855E+01
```

```
READ "(1X, A13, I3, 2X, F6.2, 3X, F10.1)", &
     STRING, K, P, R
```

  - with decimal point in data *d* is irrelevant ↑ ↑
  - with declarations

```
CHARACTER(LEN=20) :: STRING
INTEGER :: K
REAL :: P, R
```

  - **STRING** is **"The result of ! ! ! ! ! ! "**
    K is 25, **P** is 3.142 and **R** is 78.55

General forms of READ/WRITE

READ (*unit*, *format*[, **IOSTAT=***int-var*])

WRITE (unit, format[, **ADVANCE="NO"**])

unit = **INTEGER** constant, variable or expression
is a file or device reference
= * standard input/output (keyboard/screen)

format = *format list* as **CHARACTER** variable or constant
= * for list-directed input/output

OPTIONAL

IOSTAT = an **INTEGER** variable (must be declared) which returns an error status.
0 = OK, > 0 error, < 0 End of file (on **READ**)

ADVANCE="NO"
Do not move to next line at end of output record.

General forms of READ/WRITE (*cont.*)

- The status check is optional
  - but very useful for **`READ`** as it can detect errors such as letters when only numerical data are expected
  - also end of file when reading from a disk file.
    - (Can also be used with WRITE.)

- **`ADVANCE="NO"`**
  - useful for prompts so that the response can be on the same line.
  - Cannot be used with list-directed format, use "(A)" instead. (May not work with all computer/compiler combinations.)

- For other options see textbooks.

General forms of READ/WRITE (*cont.*)

- *e.g.* To check for sensible input

```
INTEGER :: IOCHECK
REAL :: TEMP
DO
   WRITE (*, "(A)", ADVANCE="NO") &
         "Enter a real number: "
   READ (*,*,IOSTAT=IOCHECK) TEMP
   IF (IOCHECK == 0) THEN
      EXIT     Leave loop when input form correct
   ELSE IF (IOCHECK > 0) THEN
      PRINT *, "Enter a NUMBER, please!"
   END IF
END DO          Repeat loop if input illegal, e.g. a letter
[Other statements]
```

General forms of READ/WRITE (*cont.*)

- *e.g.* To check for end of file

```
INTEGER :: LU      Logical unit number for file
INTEGER :: IOCHECK
REAL :: TEMP
  :
READ (LU,"(F6.2)",IOSTAT=IOCHECK) TEMP
IF (IOCHECK > 0) THEN
   PRINT *, "Input error"  Give information to show
   STOP                    where, e.g. line in file
ELSE IF (IOCHECK < 0) THEN
   PRINT *, "End of file"
   STOP
END IF
[Other statements]
```

- Always use IOSTAT when reading from a disk file.

Files

- Large (or small) volumes of data can be held in a disk file which contains many lines of information.
  - Each line of information is called a record.
  - A formatted file can be created using an editor (just as is a source file for a Fortran program).
  - A file may be created as output from a Fortran or any other program.
  - Records are read or written sequentially.
  - Files are referenced by logical units.
- Files may also hold unformatted data (in internal computer format).
- Files may also be 'Direct Access' (to any record).
  - See textbooks for full details of file access.

Files (*cont.*)

- We consider only formatted, sequential access files.
  - To connect a file to a logical unit it must be **OPEN**ed
  - **OPEN(** *unit*, **FILE=***character expression*, **IOSTAT=***int-var*, **STATUS=***char-string***)**
  - *logical unit* = positive integer constant, variable or expression.
    the range is compiler-dependent but 1 to 99 should be OK.
    Do not use *****, **5** (normally standard input) or **6** (standard output).
  - **FILE =** character variable, expression or constant
    File name, e.g. **"A:\MYDATA.DAT"**, **"mydata"** or **"dir/mydata"**
    Name will depend on computer and operating system.
  - **IOSTAT =** an INTEGER variable
    0 = OK, > 0 Error, e.g. file doesn't exist.
  - **STATUS = 'OLD'** file exists (for reading)
    = 'NEW' create it (for writing)
- See textbooks for full range of **OPEN** options.

Files (*cont.*)

- **`CLOSE(`** *unit* **`)`**
  Closes the file associated with the unit and frees the unit number. Not essential. Files are automatically closed when program ends.

- **`BACKSPACE(`** *unit* **`)`** Backspaces one record.

- **`REWIND(`** *unit* **`)`** Restarts file at the beginning.

- **`INQUIRE(`** *unit* or **`FILE=`***char-expr,*
  [**`EXIST=`***logical var,* other options]**`)`**
  Find lots of information about a logical unit or file (even if the file hasn't been opened). Full details in the textbooks, but the **`EXIST=`** option is one of the most useful, to check if a file exists before trying to open it.