

# Lecture III: Input/output

# Outline

---

- ▶ Input / output formatting
  - ▶ Internal I/O & command-line parsing
  - ▶ Opening, writing, reading, closing files
    - ▶ Formatted vs. binary I/O
    - ▶ Stream I/O
  - ▶ Asynchronous I/O
-

# Input/Output formatting

---

- ▶ To prettify output and to make it human-readable, we use `FORMAT` descriptors in connection with the `WRITE` statement
  - ▶ Although less often used nowadays, it can also be used with `READ` to input data at fixed line positions and using predefined field lengths
  - ▶ Use either through `FORMAT` statements, `CHARACTER` variable or embedded in `READ / WRITE` `fmt` keyword
-

# Output formatting

Data type	FORMAT descriptors	Examples
Integer	Iw, Iw.m	WRITE(*,'(I5)') J WRITE(*,'(I5.3)') J WRITE(*,'(I0)') J
Real (decimal and exponential forms, auto-scaling)	Fw.d Ew.d Gw.d	WRITE(*,'(F7.4)') R WRITE(*,'(E12.3)') R WRITE(*,'(G20.13)') R
Character	A,Aw	WRITE(*,'(A)') C
Logical	Lw	WRITE(*,'(L2)') L

**w**=width of the output field, **d**=number of digits to the right of decimal point,  
**m**=minimum number of characters to be used.

Variables: Integer :: J, Real :: R, Character :: C, Logical :: L

# Output formatting: miscellaneous

---

- ▶ With complex numbers provide format for both real and imaginary parts:

```
COMPLEX :: Z
```

```
WRITE (*, '(F6.3,2X,F6.3)') Z
```

- ▶ Line break and tabbing:

```
WRITE (*, '(F6.3,/,F6.3)') X, Y
```

```
WRITE (*, '(I5,T20,I5)') I, J
```

- ▶ Dynamic sizing with I0 and G0 edit descriptors
- ▶ It is possible that an edit descriptor will be repeated a specified number of times

```
WRITE (*, '(5I8)')
```

```
WRITE (*, '(3(I5,F8.3))')
```

# Internal I/O

---

- ▶ Often it is necessary to filter out data from a given character string
  - ▶ Or to pack values into a character string
  - ▶ Fortran *internal* I/O with READ & WRITE becomes handy
  - ▶ No actual files (or channels) are involved at all
-

# Internal I/O examples

---

- ▶ Extract a number from a given character string

```
CHARACTER(LEN=13) :: CL = 'Time step# 10'
```

```
INTEGER :: ISTEP
```

```
READ(CL,fmt='(10X,I3)') ISTEP
```

- ▶ Write data to a character string

```
INTEGER :: njobs
```

```
CHARACTER(LEN=60) :: CL
```

```
WRITE(CL,'(A,I0)') 'The number of jobs completed = ', njobs
```

---

# Command line input

---

- ▶ In many cases, it is convenient to give parameters for the program directly during program launch
    - ▶ Instead of using a parser, reading from an input file etc.
  - ▶ Fortran way for this is
    - ▶ `COMMAND_ARGUMENT_COUNT()` : compute the number of user-provided arguments
    - ▶ `GET_COMMAND_ARGUMENT(integer i, character arg(i))` extract the argument from position i
    - ▶ You will need internal I/O to convert e.g. integer-valued arguments into values of integer variables
-



# Command line input

---

- ▶ Example: reading in two integer values from the command line
- ▶ The (full) program should be launched as (e.g.)

```
% ./a.out 100 100
```

```
subroutine read_command_line(height, width)
  integer, intent(out) :: height, width
  character(len=10) :: args(2)
  integer :: n_args, i
  n_args = command_argument_count()
  if ( n_args /= 2 ) then
    write(*,*) ' Usage : ./exe height width'
    call abort()
  end if
  do i = 1, 2
    call get_command_argument(i,args(i))
    args(i) = trim(adjustl(args(i)))
  end do
  read(args(1),*) height
  read(args(2),*) width
end subroutine read_command_line
```

# Opening & closing files

---

- ▶ Writing to or reading from a file is similar to writing onto a terminal screen (\*) or reading from a keyboard, but
    - ▶ File must be opened with an `OPEN` statement, in which the unit number and (optionally) the file name are given
    - ▶ Subsequent writes (or reads) must refer to the given unit number
    - ▶ File should be closed at the end
-

# Opening & closing files

---

- ▶ The syntax is (the brackets [ ] indicate optional keywords or arguments)

```
OPEN([unit=]iu, file='name' [, options])  
CLOSE([unit=]iu [, options])
```

- ▶ For example

```
OPEN(10, file= 'output.dat', status='new')  
CLOSE(unit=10, status='keep')
```

---

# Opening & closing files

---

- ▶ The first parameter is the unit number
  - ▶ The keyword `unit=` can be omitted
  - ▶ The unit numbers 0, 5 and 6 are predefined
    - ▶ 0 is output for standard (system) error messages
    - ▶ 5 is for standard (user) input
    - ▶ 6 is for standard (user) output
    - ▶ These units are opened by default and should not be re-opened nor closed by the user
-

# Opening & closing files

---

- ▶ The *default* input/output unit can be referred with a star:  
WRITE(\*, ...)  
READ(\*, ...)
    - ▶ Note that these are *not* necessarily the same as the stdout and stdin unit numbers 6 and 5
  - ▶ If the file name is omitted in the OPEN, the file name will be based on unit number being opened, e.g. for unit=12 this usually means the filename 'fort.12'
-

# File opening options

---

- ▶ *status* : existence of the file
    - ▶ 'old', 'new', 'replace', 'scratch', 'unknown'
  - ▶ *position* : offset, where to start writing
    - ▶ 'append'
  - ▶ *action* : file operation mode
    - ▶ 'write', 'read', 'readwrite'
  - ▶ *form* : text or binary file
    - ▶ 'formatted', 'unformatted'
-

# File opening options

---

- ▶ *access* : direct or sequential file access
    - ▶ 'direct', 'sequential', 'stream',
  - ▶ *iostat* : error indicator, (output) integer
    - ▶ Non-zero only upon an error
  - ▶ *asynchronous*: whether to try or not asynchronous I/O
    - ▶ 'yes' or 'no'
  - ▶ *err* : the label number to jump upon error
  - ▶ *recl* : record length, (input) integer
    - ▶ For direct access files only
    - ▶ Warning (check): may be in bytes or words
-

# File opening: file properties

---

- ▶ Use INQUIRE statement to find out information about
  - ▶ file existence
  - ▶ file unit open status
  - ▶ various file attributes
- ▶ The syntax has two forms, one based on file name, the other for unit number

```
INQUIRE(file='name', options ...)  
INQUIRE(unit=iu, options ...)
```



# File opening: file properties

---

- ▶ exist : does file exist ? (LOGICAL)
  - ▶ opened : is file / unit opened ? (LOGICAL)
  - ▶ form : 'formatted' or 'unformatted' (CHAR)
  - ▶ access : 'sequential' or 'direct' or 'stream' (CHAR)
  - ▶ action : 'read', 'write', 'readwrite' (CHAR)
  - ▶ recl : record length (INTEGER)
  - ▶ size : file size in bytes (INTEGER)
-

# File opening: file properties

---

- ▶ Find out about existence of a file

```
LOGICAL :: file_exist
```

```
INQUIRE (FILE='foo.dat', EXIST=file_exist)
```

```
IF (.NOT. file_exist) THEN
```

```
    WRITE(*,*) 'The file does not exist'
```

```
ELSE
```

```
    ! Do something with the file 'foo.dat'
```

```
ENDIF
```

---

# File writing and reading

---

- ▶ Writing to and reading from a file is done by giving the corresponding unit number (iu) as a parameter :

```
WRITE(iu,*) str
```

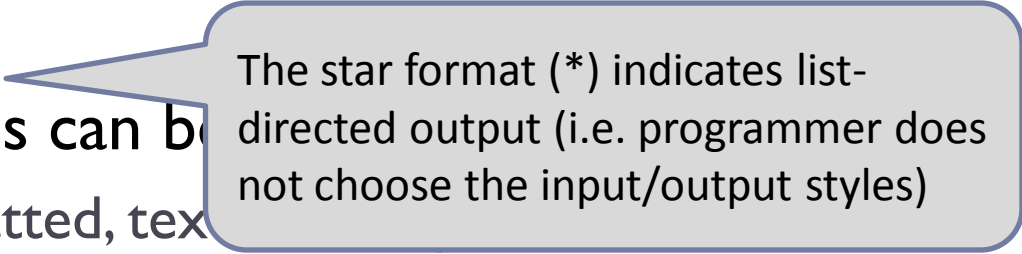
```
WRITE(unit=iu, fmt=*) str
```

```
READ(iu,*) str
```

```
READ(unit=iu, fmt=*) str
```

- ▶ Formats and other options can be

- ▶ 'fmt' is applicable to formatted, text



The star format (\*) indicates list-directed output (i.e. programmer does not choose the input/output styles)

# Formatted vs. unformatted files

---

- ▶ Text or *formatted* files are
    - ▶ Human readable
    - ▶ Portable i.e. machine independent
  - ▶ Binary or *unformatted* files are
    - ▶ Machine readable only, *not* portable
    - ▶ Much *faster to access* than formatted files
    - ▶ Suitable for large amount of data due to *reduced file sizes*
    - ▶ Internal data representation used for numbers, thus no number conversion, no rounding of errors compared to formatted data
-

# Unformatted I/O

---

- ▶ Write to a sequential binary file

```
REAL rval
```

```
CHARACTER(len = 60) string
```

```
OPEN(10,file='foo.dat',form='unformatted')
```

```
WRITE(10) rval
```

```
WRITE(10) string
```

```
CLOSE(10)
```

- ▶ No FORMAT descriptors allowed

- ▶ Reading similarly

```
READ(10) rval
```

```
READ(10) string
```

---

# Stream I/O

---

- ▶ A binary file write adds extra record delimiters (hidden from programmer) to the beginning and end of records
  - ▶ In Fortran 2003 using access method 'stream' avoids this and implements a C-like approach
    - ▶ One should move to use stream I/O for efficiency and portability
  - ▶ Create a stream (binary) file  
`OPEN(10,file='my_data.dat',access='stream')`
-

# Asynchronous I/O

---

- ▶ Both writing and reading can be *asynchronous* i.e. other statements being executed while I/O in progress
  - ▶ It is system dependent whether I/O is asynchronous for real

```
OPEN(10, ..., asynchronous='yes')  
WRITE(10,..., id=id, asynchronous='yes') A  
CALL do_something(....) ! Not involving A here  
WAIT(10, id=id) ! Blocks here until A has been written  
CALL do_something(...) ! OK to use A here
```

# Asynchronous I/O

---

- ▶ If the asynchronous file access is performed in a procedure other than the one called for OPEN, the data involved has to be declared with "asynchronous" attribute

```
OPEN(10, ..., asynchronous='yes')
CALL async_write(10, A, id)
CALL do_something_else_here()
WAIT(10, id=id)
...
SUBROUTINE async_write(iu, data, id)
  INTEGER, INTENT(IN) :: iu
  INTEGER, INTENT(IN), DIMENSION(:), ASYNCHRONOUS :: data
  INTEGER, INTENT(OUT) :: id
  ...
  WRITE(iu, id=id, asynchronous='yes') data
  ...
END SUBROUTINE async_write
```

---



# Asynchronous I/O

---

- ▶ An alternative for calling `WAIT` is to periodically call `INQUIRE` to check the status of the operation and in the meantime keep on doing something else
- ▶ Not necessarily supported in all platforms but the `INQUIRE` statement equals to `WAIT`

```
LOGICAL :: status
```

```
...
```

```
OPEN(10, ..., asynchronous='yes')
```

```
WRITE(10,..., id=id, asynchronous='yes') A
```

```
DO WHILE (!status)
```

```
    CALL do_something(....) ! Not involving A
```

```
    INQUIRE (10, id=id, pending=status)
```

```
END DO
```

---

# Summary

---

- ▶ Input/Output formatting
  - ▶ Files: communication between a program and the outside world
    - ▶ Opening and closing a file
    - ▶ Data reading & writing
  - ▶ Use unformatted (binary) I/O for all except text files
    - ▶ Stream I/O
  - ▶ Asynchronous I/O may be beneficial with large datasets
  - ▶ Internal I/O & command-line parsing
-

# Programming assignment

---

The files `gol.f95` and `gol_io.f95` contain an implementation of the "Game of Life", a famous cellular automaton, read

[http://en.wikipedia.org/wiki/Conway's\\_Game\\_of\\_Life](http://en.wikipedia.org/wiki/Conway's_Game_of_Life)

The I/O module in `gol_io.f95` is missing the code for writing the GoL boards into image files (here in `.pbm` format) and for reading in the command-line input. Complete the missing pieces of code.

For how to write `.pbm` images, see

[http://en.wikipedia.org/wiki/Netpbm\\_format](http://en.wikipedia.org/wiki/Netpbm_format) Feel free to implement it in some other picture format!

Complete the code and run it to see how the automaton evolves. Return the completed code together with some sample output (images/animations).

---