

Input and Output

Fortran I/O Overview

- Input/output (I/O) can be a lot more flexible than just reading typed input from the terminal window and printing it back out to a screen.
- Fortran allows for **multiple file streams**.
- Fortran allow **multiple representations** of the data for I/O.
- Fortran allows multiple approaches to the **sequencing** of I/O.

Some More I/O definitions

- **File** - a collection of data
- Data is organized into **records**, which may be **formatted** (character representation), **unformatted** (machine binary representation), or denote an **end of file**. (Compare: a Unix file is a sequence of bytes.)
- Each **READ** and **WRITE** uses **I+ records**. Any unread characters are skipped for **READ**. **WRITE** ends by writing an **end-of-line** indicator.
- (**direct access** is an exception to the above)

Really Basic I/O (again)

`READ *, <variable list>` reads from `stdin`

`PRINT *, <expression list>` writes to `stdout`

`WRITE(*,*), <expression list>` writes to `stdout`

Both do input/output as **human-readable text**.

Each I/O **statement** reads/writes on a new line.

Input data can be on single line or multiple lines,
comma or space delimited (if the read requires
multiple numbers).

Character (string) input must be put in quotes.

See class 8, example1.f90 with stdin files `stdin.1`,
`stdin.2` and `stdin.3`.

Formatting

READ <format>, <variable list>

PRINT <format>, <expression list>

WRITE(*, <format>), <expression list>

The **format** specifier is used in **read**, **write** and **print** statements

* - default, or list-directed formatting

f (floating point) for I O of reals

syntax: '(**f****w.****d**)' where

w = total number of positions

d = number of places after the decimal point

The decimal point occupies a position, as does the minus sign

Formatting (cont.)

e (exponential) for I O of large and small reals

syntax: '(ew.d)' where

w = total number of positions

d = number of digits in mantissa

a (alphanumeric) for character strings

syntax: '(aw)' where

w = total number of positions

i (integer) for character strings

syntax: '(iw)' or '(iw.d)' where

w = total number of positions

d = the number of zeros that will pad the value

Any format can be repeated with a leading number
and can be mixed and matched.

Formatting examples

/ for newline

‘(f12.2)’

‘(e12.4)’

‘(i2)’

‘(i4.4)’

‘(f8.1/2e13.5)’

‘(3(a12,4i6))’

Be sure the format is sized to represent the number you expect.

A floating point format requires $W \geq D+3$

An exponential format requires $W \geq D+7$

IOSTAT Keyword

The **IOSTAT** keyword lets you test for various error conditions associated with in I O operation. Zero is returned for an operation that completes normally. The meaning of other values is compiler dependent. One can test for specific conditions such as end-of-file or end-of-record.

```
DO  
  READ(*,*,IOSTAT=ierr) x  
  IF(ierr /= 0) EXIT  
  
  ...  
ENDDO
```

When the **IOSTAT** keyword is omitted you get an execution error for abnormal conditions. With **IOSTAT** it returns to you the code and continues onward.

Multiple File Streams

A keyword nearly universal to all Fortran I/O statements is the **Logical Unit**

`WRITE(11,*)`u !written to file associated with unit 11
`WRITE(12,*)`v !written to file associated with unit 12

```
INTEGER :: lun=3
READ(lun,*)n
```

Default filename associated with logical unit **lun** is **fort.lun** (`fort.11`, `fort.12`, `fort.3`). **Compilers may vary!** Generally use 1 through 99.

Open Statement

The OPEN statement associates a logical unit with a specific file:

```
OPEN( [UNIT=],<integer>,FILE=<char>, &
      FORM=<char>,ACCESS=<char>, &
      ACTION=<char>,STATUS=<char>, &
      POSITION=<char>,RECL=<integer>, &
      IOSTAT=<integer var>)
```

```
OPEN(UNIT=10,FILE='input.u',FORM='formatted')
OPEN(21,FILE='output.dat',FORM='unformatted', &
      STATUS='OLD',ACTION='READ')
```

Open Statement (cont.)

More on common **OPEN** keywords:

FORM: ‘**FORMATTED**’ or ‘**UNFORMATTED**’

ACCESS: ‘**SEQUENTIAL**’ (default) or ‘**DIRECT**’

POSITION: ‘**ASIS**’ (default), ‘**REWIND**’ or ‘**APPEND**’

ACTION: ‘**READWRITE**’ (default), ‘**READ**’ or ‘**WRITE**’

STATUS: ‘**UNKNOWN**’ (default), ‘**OLD**’, ‘**NEW**’,
‘**REPLACE**’ or ‘**SCRATCH**’

RECL: integer record length for direct access I/O

One can open an already connected file to change its properties.

CLOSE Statement

The CLOSE statement terminates the connection of a file to a logical unit. A normal program exit will automatically do this.

```
Close( [UNIT=],<integer>, STATUS=<char>, &  
      IOSTAT=<integer var>)
```

STATUS: what to do with the closed file - 'KEEP' (default) or 'DELETE'.

More READ and WRITE

```
READ( [UNIT=]<integer>, [FMT=]<format>, &
      END=<label>, ERR=<label>, REC=<integer>,&
      ADVANCE=<char>, IOSTAT=<integer var>)
```

```
WRITE( [UNIT=]<integer>, [FMT=]<format>, &
       END=<label>, ERR=<label>, REC=<integer>,&
       ADVANCE=<char>, IOSTAT=<integer var>)
```

ADVANCE: ‘YES’ (default) or ‘NO’

REC: the record number in direct access I/O

END and **ERR** obsolescent - use **IOSTAT**

INQUIRE

The **INQUIRE** statement can get information about a file. You may inquire by **UNIT** or by **FILENAME**.

```
INQUIRE( [UNIT=]<integer>, EXIST=<logical_var>, &
          NAME=<char_var>, OPENED=<logical_var> &
          IOSTAT=<integer var>)
```

```
INQUIRE( [NAME=]<char_var>, EXIST=<logical_var>, &
          UNIT=<integer>, OPENED=<logical_var> &
          IOSTAT=<integer var>)
```

plus many more arguments.

Other useful statements

The following are position statements and let you change your position within a sequential access file:

REWIND ([UNIT=]<integer>,IOSTAT=<integer var>)

**BACKSPACE ([UNIT=]<integer>, &
IOSTAT=<integer var>)**

ENDFILE ([UNIT=]<integer>,IOSTAT=<integer var>)

Example programs **avg1.f90**, **avg2.f90**, **avg3.f90** and
avg4.f90 demonstrate some features of I/O.

Unformatted I/O

When a file is opened with
FORM='UNFORMATTED' the data will be read/
written in the machine binary representation. Use no
format specifier!

Unformatted I/O is much faster, more compact.

Warning! Different machines may have different
representations - big-endian vs. little-endian; latter is
prevalent nowadays.

Sequential Access

Sequential Access (the default) advances record by record through the file. The end of each record is marked by a special signifier.

As name implies, each **READ/WRITE** proceeds to the next record - exception is when **ADVANCE='NO'** is used.

Can control file position with **POSITION** statements.

Direct Access

Permits user to specify exactly which bytes are addressed in a file by an I/O operation - no end of record markers. Multiple jobs/processes can access the file without interference.

Must open file with **ACCESS='DIRECT'** and specify a record length **RECL=<integer>** (generally in bytes)

You go directly where you wish in the file by specifying the record number **REC=<integer>** in the **READ/WRITE**

NAMELIST

NAMELIST I/O is a deprecated type of formatted I/O

LOGICAL:: dopbp

INTEGER :: ijtlen

NAMELIST /pbplist/ dopbp, ijtlen

OPEN(2,FILE='namel.pbp',FORM='FORMATTED'

READ(2,pbplist)

> cat namel.pbp

&pbplist

dopbp=.true.

ijtlen=4

&END

Internal I/O

Imagine you wish to convert a number to its character representation:

```
CHARACTER (LEN=4):: cyear  
INTEGER, PARAMETER :: year = 1989  
OPEN(2,FILE='temfile',FORM='FORMATTED')  
WRITE(2,FMT='(i4)')year  
BACKSPACE(2)  
READ(2,FMT='(A4)')cyear
```

Internal I/O does this directly where the logical unit is a variable rather than a file

```
WRITE(UNIT=cyear,FMT='(i4)')year  
READ(UNIT=cyear,FMT='(i4)')newyear
```

I O Libraries

Typically, with standard fortran I O statements when someone sends you a file he must also send you a README about the contents, or some code kernal for reading

It sure would be nice if the data in files were ‘self-describing’ with the use of ‘metadata’!

I O libraries are publicly available that can do this: NetCDF and HDF are widely used in atmospheric sciences.

NetCDF

NetCDF is something of a standard for climate/
meteorological data:

<http://www.unidata.ucar.edu/software/netcdf>

Includes command line utilities to inspect the files

Many graphics packages can read it (IDL)

NCO (<http://nco.sourceforge.net>) is a set of
command line utilities to manipulate NetCDF files

Fortran subroutine calls are used to read/write/
inquire about the data.

Examples: `sfc_pres_temp_wr.f90`,
`sfc pres temp rd.g90`

A Digression on Libraries

Libraries are code that have already been compiled.
You need to tell your program where to find them.

-I<path_to_include_files_and_modules>

-L<path_to_libraries> -l<library_name>

To compile one of the netcdf examples:

```
ifort -I/usr/local/intel/include sfc_pres_temp_rd.f90 \
```

```
      -L/usr/local/intel/lib -lncdf
```

Other examples of libraries for linear algebra (lapack),
pde solvers(phaml), fft (fftw) and many more.