Chapter 11
Summary of I/O Concepts

It is a capital mistake to theorise before one has data
Sir Arthur Conan Doyle

Aims
This chapter covers more formally some of the concepts introduced in Chaps. 9 and 10. There is a coverage of

- I/O concepts and I/O statements
- Files, records and streams
- Sequential, direct and stream access
- Options or specifiers on the `open` statement
- Options or specifiers on the `close` statement
- Options or specifiers on the `write` statement
- Options or specifiers on the `read` statement

11.1 I/O Concepts and Statements

Fortran input and output statements provide the means of transferring data from external media to internal storage or from an internal file to internal storage and vice versa.

The input/output statements are the `open`, `close`, `read`, `write`, `print`, `backspace`, `endfile`, `rewind`, `flush`, `wait`, and `inquire` statements.

The `inquire` statement is a file inquiry statement.

The `backspace`, `endfile`, and `rewind` statements are file positioning statements.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran*,
https://doi.org/10.1007/978-3-319-75502-1_11

Data is commonly organised in either record files or stream files. In a record type file transfers are done a record at a time. In a stream type file transfers are done in file storage units.

11.2 Records

A record is a sequence of values or a sequence of characters. There are three kinds of records:

- formatted
- unformatted
- end of file

A record in Fortran is commonly called a logical record.

A formatted record is typically a sequence of printable characters. You have seen examples in earlier chapters.

You saw examples of unformatted i/o in the previous chapters.

11.3 File Access

The three file access methods are:

- sequential
- direct
- stream

The examples so far have shown sequential access.

Direct access is a method of accessing the records of an external record file in arbitrary order.

Stream access is a method of accessing the file storage units of an external stream file. The properties of an external file connected for stream access depend on whether the connection is for unformatted or formatted access.

11.4 The `open` Statement

An `open` statement initiates or modifies the connection between an external file and a specified unit. The `open` statement can do a number of things including

- connect an existing file to a unit;
- create a file that is preconnected;

- create a file and connect it to a unit;
- change certain modes of a connection between a file and a unit.

The only keyword option that can be omitted is the `unit` specifier. This is assumed to be the first parameter of the `open` statement.

Table 11.1 summarises the `open` statement options.

Table 11.1 Open statement options

unit =	file-unit-number
access =	sequential, direct or stream
action =	read, write or readwrite
asynchronous =	yes or no
blank =	null or zero
decimal =	comma or point
delim =	apostrophe, quote or none
encoding =	utf8 or default
err =	statement label
file =	file name
form =	formatted or unformatted
iomsg =	iomsg-variable
iostat =	scalar-int-variable
newunit =	scalar-int-variable
pad =	yes or no
position =	asis, rewind, append
recl =	record length, positive integer
round =	up, down, zero, neareset, compatible or processor defined
sign =	plus, suppress or processor defined
status =	old, new, scratch, replace or unknown

11.5 Data Transfer Statements

The `read`, `write` and `print` statements are used to transfer data to and from files.

Table 11.2 summarises the options of the data transfer statements.

Table 11.2 Data transer statement options

unit =	io-unit
fmt =	format
nml =	namelist-group-name
advance =	yes or no
asynchronous =	yes or no
blank =	null or zero
decimal =	comma or point
delim =	apostrophe, quote or none
end =	label
eor =	label
err =	label
id =	scalar-int-variable
iomsg =	iomsg-variable
iostat =	scalar-int-variable
pad =	yes or no
pos =	file position in file storage units
rec =	record number to be read or written
round =	up, down, zero, neareset, compatible or processor defined
sign =	plus, suppress or processor defined
size =	scalar-int-variable

11.6 The `inquire` Statement

Table 11.3 summarises the options on the `inquire` statement.

Table 11.3 Inquire statement options

unit =	file-unit-number
file =	file name
access =	sequential, direct, stream
action =	read, write, readwrite, undefined
asynchronous =	yes, no
blank =	zero, null
decimal =	comma, point
delim =	apostrophe, quote, none
direct =	yes, no, unknown

Table 11.4 (continued)

encoding =	utf8, default
err =	label
exist =	true, false
form =	formatted, unformatted, undefined
formatted =	yes, no, unknown
id =	scalar-int-expr
iomsg =	iomsg-variable
iostat =	scalar-int-variable
name =	file name
named =	scalar-logical-variable
nextrec =	scalar-int-variable
number =	unit number, -1 if unassigned
opened =	true, false
pad =	yes, no
pending =	scalar-logical-variable
pos =	scalar-int-variable
position =	scalar-default-char-variable
read =	yes, no, unknown
readwrite =	yes, no, unknown
recl =	scalar-int-variable
round =	up, down, zero, neareset, compatible or processor defined
sequential =	yes, no, unknown
sign =	plus, suppress, processor defined
size =	scalar-int-variable
stream =	yes, no, unknown
unformatted =	yes, no, unknown
write =	yes, no, unknown

11.7 Error, End of Record and End of File

The set of input/output error conditions is processor dependent.

An end-of-record condition occurs when a non-advancing input statement attempts to transfer data from a position beyond the end of the current record, unless the file is a stream file and the current record is at the end of the file (an end-of-file condition occurs instead). An end-of-file condition occurs when

- an endfile record is encountered during the reading of a file connected for sequential access,
- an attempt is made to read a record beyond the end of an internal file, or
- an attempt is made to read beyond the end of a stream file.

An end-of-file condition may occur at the beginning of execution of an input statement. An end-of-file condition also may occur during execution of a formatted input statement when more than one record is required by the interaction of the input list and the format. An end-of-file condition also may occur during execution of a stream input statement.

11.7.1 Error Conditions and the `err=` Specifier

The set of error conditions which are detected is processor dependent. The standard does not specify any i/o errors. Compilers will vary in the errors they detect and how they treat them. The `err=` option provides one way of catching errors and taking the appropriate action.

11.7.2 End-of-File Condition and the `end=` Specifier

An end of file may occur during an input transfer. The `end=` option provides a way of handling the end of file in a program.

11.7.3 End-of-Record Condition and the `eor=` Specifier

An end of record may occur during an input transfer. The `eor=` option provides a way of handling this in a program.

11.7.4 `iostat=` Specifier

Execution of an input/output statement containing the `iostat=` specifier causes the scalar-int-variable in the `iostat=` specifier to become defined with one of a set of values. Normally

- 0 if no errors occur
- a processor dependent negative value if end-of-file occurs
- a processor dependent negative value if an end-of-record occurs

If you use `iostat_inquire_internal_unit` from the intrinsic module `iso_fortran_env` you will get a processor-dependent positive integer value if a unit number in an inquire statement identifies an internal file.

When using `iostat_inquire_internal_unit` you will get a processor-dependent positive integer value which is different from the above if any other error condition occurs,

11.7.5 `iomsg=` Specifier

If an error, end-of-file, or end-of-record condition occurs during execution of an input/output statement, the processor shall assign an explanatory message to `iomsg-variable`. If no such condition occurs, the processor shall not change the value of `iomsg-variable`.

11.8 Examples

Here are three examples using the `iostat=` option. Examples illustrating some of the other options can be found throughout the rest of the book.

11.8.1 Example 1: Simple Use of the `read`, `write`, `open`, `close`, `unit` Features

This example shows the use of several of the i/o features including

- the `write` statement
- the `read` statement
- the use of `unit=6` on a write statement
- the use of `unit=5` on a read statement
- several `fmt=` variations
- the `open` statement
- the `file=` option on the `open` statement
- the `iostat=` option on the `open` statement
- the `close` statement

```
program ch1101
  implicit none
  integer :: filestat
  real :: x
  character (len=20) :: which

 do
```

```
    write (unit=6, fmt= &
      '("data file name,or end")')
    read (unit=5, fmt='(a)') which
    if (which=='end') exit
    open (unit=1, file=which, iostat=filestat, &
      status='old')
    if (filestat>0) then
      print *, &
        'error opening file, please check'
      stop
    end if
    read (unit=1, fmt=100) x
    write (unit=6, fmt=110) which, x
    close (unit=1)

  end do

100 format (f6.0)
110 format ('from file ', a, ' x = ', f8.2)
end program ch1101
```

It is common for compilers to associate units 5 and 6 with the keyboard and screen.

11.8.2 Example 2: Using `iostat` to Test for Errors

```
program ch1102
  implicit none
  integer :: io_stat_number = -1
  integer :: i

  do
    print *, 'input integer i:'
    read (unit=*, fmt=100, iostat=io_stat_number &
      ) i
    print *, ' iostat=', io_stat_number
    if (io_stat_number==0) exit
  end do
  print *, 'i = ', i, ' read successfully'
100 format (i3)
end program ch1102
```

11.8.3 Example 3: Use of `newunit` and `lentrim`

This example illustrates the use of the following:

- the `len_trim` function
- the `newunit` option on the `read` statement to get an unused `unit` number
- the use of `iostat=` to test whether a file was opened correctly
- the use of the `cycle` control statement to go back to the start of the `do` and try reading the file name again
- the use of the `iostat` option to test if the read was successful

```
program ch1103
  implicit none
  character (len=20) :: station, file_name
  integer :: i, io_stat_number, filestat, flen, &
    uno
  integer, parameter :: nmonths = 12
  integer, dimension (1:nmonths) :: year, month
  real, dimension (1:nmonths) :: rainfall, &
    sunshine
  real :: rain_sum
  real :: rain_average
  real :: sun_sum
  real :: sun_average

  do
    print *, 'input weather station'
    print *, ' or "end" to stop program'
    read '(a)', station
    if (station=='end') exit
    flen = len_trim(station)
    file_name = station(1:flen) // 'data.txt'
    open (newunit=uno, file=file_name, &
      iostat=filestat, status='old')
    if (filestat/=0) then
      print *, 'error opening file ', file_name
      print *, 'Retype the file name'
      cycle
    end if
    do i = 1, 7
      read (unit=uno, fmt='(a)')
    end do
    do i = 1, nmonths
      read (unit=uno, fmt=100, iostat= &
        io_stat_number) year(i), month(i), &
        rainfall(i), sunshine(i)
100   format (3x, i4, 2x, i2, 27x, f4.1, 3x, &
        f5.1)
      if (io_stat_number/=0) then
        print *, ' error reading record ', &
          i + 8, &
          ' so following results incorrect:'
        exit
      end if
    end do
    close (unit=uno)
    rain_sum = sum(rainfall)/25.4
    sun_sum = sum(sunshine)
    rain_average = rain_sum/nmonths
    sun_average = sun_sum/nmonths
    write (unit=*, fmt=110) station
110 format (/, /, 'Station = ', a, /)
    write (unit=*, fmt=120) year(1), month(1)
120 format (2x, 'Start ', i4, 2x, i2)
    write (unit=*, fmt=130) year(12), month(12)
130 format (2x, 'End   ', i4, 2x, i2)
    write (unit=*, fmt=140)
140 format (19x, ' Yearly Monthly', /, 19x, &
      ' Sum Average')
    write (unit=*, fmt=150) rain_sum, &
      rain_average
150 format ('Rainfall (inches) ', f7.2, 2x, &
      f7.2)
    write (unit=*, fmt=160) sun_sum, sun_average
160 format ('Sunshine ', f7.2, 2x, f7.2)
  end do
end program ch1103
```

In this program based on an earlier example in Chap. 10, we have use of the `newunit` option on the `open` statement. A unique negative number is returned, which cannot clash with any user specified unit number, which are always positive.We are also using the character intrinsic function `len_trim` and the character operator

//

We also introduce the `do end do` and `cycle` statements. These are covered in more detail in Chap. 13.

11.9 Unit Numbering

Care must be taken with unit numbering as firstly they must always be positive, and secondly many compilers have conventions that apply, for example unit 5 is often associated with the `read *` statement and unit 6 is often associated with the `print *` statement.

11.10 Summary

This chapter has listed most of the i/o options available in Fortran. There are a small number of examples that illustrate some of their use.

Later chapters provide additional examples.

11.11 Problems

The Whitby data and Cardiff data are on our web pages.

11.1 Compile and run the examples in this chapter.

11.2 With the Whitby or Cardiff data make a mistake, e.g. a non-numeric character in the last column. Test program `ch1103.f90` to see that it picks this up.