Introduction to
Scientific & Engineering Computing
BIL 102FE (Fortran) Course
for
Week 9

Dr. Ali Can Takinacı

Assistant Professor

in

The Faculty of Naval Architecture and Ocean Engineering

80626

Maslak – Istanbul – Turkey

MORE CONTROL OVER INPUT AND OUTPUT

The input and output facilities of any programming language are extremely important, because they establishes communication between the user and the program is carried out.

The list-directed input and output statements, which are used up to now, provide the capability for straightforward input from the keyboard and output to the display or printer.

These statements allow the user very little control over the source or the layout of the input data or over the destination or layout of the printed results.

This chapter introduces the more general input and output features of F, by means of which the programmer may specify exactly how the data will be presented and interpreted, from which of the available input units it is to be read, exactly how the results are to be sent.

Format and edit descriptors

Up to this point all the input and output has been carried out using list-directed `read` and `print` statements. They are restricted in their ability to define both the format of the information, especially, its source or destination.

An input statement must contain three distinct types of information:

1. where the data is to be found
2. where it is to be stored in the computer's memory
3. how it is to be interpreted

The key element in both input and output process is the editing of information in one form presentation in another form. The input and output statements have taken the forms up to now.

```
read *, input_list
print *, output_list
```

But each of these statements also has an alternative form:

```
read chr_expr, input_list
print chr_expr, output_list
```

where *chr_expr* is a character expression.

In both forms the item following the keyboard (`read` or `print`) is a **format specifier**

- This provides a link to the information necessary for the required editing to be carried out as part of the input or output process.

- This information is called a **format** and consists of a list of **edit descriptors** enclosed in parentheses.

- `(ed_des1, ed_des2, ...)`

In the list-directed form the asterisk (*) indicated that the format to be used is

a list-directed format which will be created by the processor to meet the perceived needs of the particular input or output list.

– In the new form, the format is called an **embedded format** because it appears as part of the `read` or `print` statement.

– `read “(`*`edit_descriptor_list`*`)”, `*`input_list`*

It is also possible to store such a format in a character variable, and then include the name of the variable, which is the simplest form character expression, in the `read` or `print` statement:

- `print` *`print_format, output_list`*

where *`print_format`* is a character variable containing the format.

Input editing

The first and simplest edit descriptor is used for inputting whole numbers, which are to be stored in an *integer* variable, and takes the form

- `iw`

- This indicates that the next *w* characters are to be read and interpreted as an integer. Thus if one wished to read the line

- `123456789`

- as a single integer to be stored in the integer variable `n` it could be written

- `read “i9”, n`

If one wished to read the same line as three separate integers (123, 456, 789) then it would be written

- `read “i3, i3, i3”, n1, n2, n3`

where n1, n2 and n3 are integer variables.

This format interacts with the rest of the read statement in the following way:

- First the `read` statement recognises that it requires an integer to store in `n1`; the format indicates that the first item to be read is an integer occupying the first three character positions (i3). The characters "123" are therefore read and converted to the internal form of the integer 123 before being stored in `n1`.

- The `read` statement then requires another integer and the format indicates that this is to come the *next* three character positions (i3). The characters "`456`" are therefore read and converted to the internal form of the integer `456` before being stored in n2.

- Finally, the process is repeated a third time, causing the characters "`789`" to be read, converted, and stored in n3 as integer `789`.

- The read statement is satisfied, since data has been read for all of the variables in its input list, and so input of this line of data is completed.

The full list of edit descriptors is:

Descriptor	***Meaning***
`i`*`w`*	Read the next *w* characters as an integer
`f`*`w`*`.`*`d`*	Read the next *w* characters as a real number with d digits after the decimal place if no decimal point is present
`a`*`w`*	Read the next *w* characters as characters
`a`	Read sufficient characters to fill the input list item, stored as characters
`L`*`w`*	Read the next *w* characters as the representation of a logical value
`t`*`c`*	Next character to be read is at position *c.*
`t1`*`n`*	Next character to be read is *n* characters before (`t1`) of
`tr`*`n`*	after (`tr`) the current position

The next data edit descriptor is the f edit descriptor, which is used for reading real values, and takes a slightly more complicated form than that used for integers:

f*w*.*d*

If the data is typed with a decimal point in the appropriate position then the edit descriptor causes the next *w* characters to be read and converted into a real number. The value of *d* is irrelevant (although it must be included in the format).

On the other hand if the *w* columns which are to be read as a real number do not contain any decimal point then the value of *d* indicated where one may be assumed to have been omitted, by specifying that the number has *w* decimal places

The input record are `123456789`

the statement

```
read “f9.4”, real_num
```

will cause the first nine characters to be read as a real number with four decimal places.

the number `12345.6789`

```
read “f3.1, f2.2, f3.0, t16, f4.2”, r1, r2, r3, r4
```

will cause the value

`12.3` to be stored in `r1`
`0.45` in `r2`
`678.0` in `r3`
`34.56` in `r4`.

Since `t16` edit descriptor (`t1`*n*) specifies *a relative tab* – that is a move of 6 characters position to the *left*.

The current position is in `8` after execution of the `f3.0` then `t16` moves the point reading to `3`.

In Summary

```
read “f3.1, f2.2, f3.0, t16, f4.2”, r1, r2, r3, r4
```

Data:	12345678	.23.56.8
r1 contains	12.3	0.23
r2 contains	0.45	0.5
r3 contains	678.0	6.8
r4 contains	34.56	3.56

There is one further point to be made about the format of real data. The exponential format is allowed for numbers being input by a `read` statement. In this case, the exponent may take one of three forms

- a signed integer constant
- e (or E) followed by an optionally signed constant
- d (or D) followed by an optionally signed constant

In the latter two cases the letter (e, E, d or D) may be followed by one or more spaces. The interpretation is identical, regardless of which letter is used.

Thus a real data value may be written in a great many different ways; for example, some of the ways in which the number `361.764` may occur in data are shown in the figure below.

```
361.764     3.61764+2       361764-3     0.0361764d-1
3617.64d-1   3.61764e+2      361.764+0
```

The third major data edit descriptor is the `a` edit descriptor, which is used to control the editing of character data. It takes one of the form

`a`*`w`*
`a`

During input, the edit descriptor `a`*`w`* refers to the next *`w`* characters (just as `i`*`w`* `and` `f`*`w`*`.`*`d`* refer to *`w`* characters).

If one assumes that the length of the input list item is *len* then the following rules apply

- If *w* is less than *len* then extra blank characters will be added at the end so as to extend the length of the input character string to *len*. This is similar to the situation with assignment.

- If *w* is greater than *len* , the rightmost *len* characters of the input character string will be stored in the input list item.

Thus, if the three variables `ch1`, `ch2` and `ch3` are declared by the following statements:

```
character (len=10) :: ch1
character (len=6) :: ch2
character (len=15) :: ch3
```

then the following two statements will have the identical effect:

```
read “a10, a6, a15”, ch1, ch2, ch3
read “a, a, a”, ch1, ch2, ch3
```

The remaining data edit descriptor is used with logical data, and takes the form

L*w*

where it is used an upper-case L to avoid the potential confusion with the digit l that can be caused to human readers by using the lower-case form.

This edit descriptor processes the next *w* characters to derive either as *true* value, a *false* value or an error. Thus any of the following are acceptable as representing *true*:

```
t
true
.T
 .true.
truthful
```

while the following will be interpreted as *false.*

```
F
False
.f
 .true.
futile
```

Output editing

The edit descriptors used for outputs are essentially the same as those used for input, although there are some additional ones that are only available for output and the interpretation of the others is slightly different.

The edit descriptors for output

Descriptor	*Meaning*
`i`*w*	Output an integer in the next *w* character positions
`f`*w*`.`*d*	Output a real number in the next *w* character positions with `d` decimal places
`A`*w*	Output a character string in the next *w* character positions
`A`	Output a character string, starting at the next character position, with no leading or trailing blanks
`L`*w*	Output *w* – 1 blanks, followed by `T` or `F` to represent a logical value
`T`*c*	Output the next item starting at character *c*.
`t1`*n*	Output the next item starting *n* character positions before (`t1`) or
`tr`*n*	after (`tr`) the current position

The `i` edit descriptor (`i`*`w`*) causes an integer to be output in such a way as to utilise the next *w* character positions. These *w* positions will consist of one or more spaces (if necessary), followed by the value of the number. Thus the statements

```
tom = 23
dick = 715
harry = -12
print “(i5, i5, i5)”, tom, dick, harry
```

will produce the following line of output where the symbol ◊ represents a space

◊◊◊23◊◊715◊◊-12

The `f` edit descriptor operates in a similar way, and `f`*`w.d`* indicates that a real number is to be output occupying *w* characters, of which the last *d* are to follow the decimal point. Note that the real value to be output is *rounded* (nor truncated) to *d* places of decimals before it is sent to the relevant output device. Rounding is carried out in the usual arithmetic way. Thus the statements

```
x = 3.14159
y = -275.3024
z = 12.9999
 print “(f10.3, f10.3, f10.3)”, x, y ,z
```

will produce the following line of output:

```
◊◊◊◊3.142◊◊-275.302◊◊◊13.000
```

Because the edit descriptors each specify only three places of decimals, the value of x is printed as `3.142` (rounded up), the value of y as `-275.302` (rounded down), and the value of z as `13.000` (rounded up).

It is important to realise that, for all numeric edit descriptors, if the number does not require the full **field width** *w* it will be preceded by one or more spaces across the page and the printing of tables becomes relatively easy. An example of this technique is shown in the following program:

```
program tabular_output
    real, parameter :: third = 1.0 / 3.0
    real :: x
    integer :: i
    do i = 1 , 10
    x = i
!   print "(f15.4, f15.4, f15.4)", x,sqrt(x),x**third
    print "(3f15.4)", x,sqrt(x),x**third
    end do
    end program tabular_output
```

```
         1.0000         1.0000         1.0000
         2.0000         1.4142         1.2599
         3.0000         1.7321         1.4422
         4.0000         2.0000         1.5874
         5.0000         2.2361         1.7100
         6.0000         2.4495         1.8171
         7.0000         2.6458         1.9129
         8.0000         2.8284         2.0000
         9.0000         3.0000         2.0801
        10.0000         3.1623         2.1544
```

The `a` edit descriptor works in a similar fashion for output as it does for input, and `a`*`w`* will cause characters to be output to the next *`w`* character positions of the output record. As was the case for input, it needs to be establish exactly what happens if the length of the output list item is not exactly *`w`*`.` The rules that apply here are similar to that of for input:

• If *`w`* is greater than *`len`* then the character string will be right-justified within the output field, and will be preceded by one or more blanks. This is similar to what happens with the `i` and `f` edit descriptors.

• If *`w`* is less than *`len`* then the leftmost *`w`* characters will be output.

If a character string is output to a field larger than its length then it will have spaces added at the beginning of data.

Finally, there is the `L` edit descriptor for use in outputting a representation of logical values. This is perfectly straightforward, and the descriptor `Lw` will cause `w-1` blanks to be output, followed by the letter `T` or the letter `F` to indicate `true` or `false`.

There is one further point that should be made at this point. A number, called **a repeat count**, may be placed *before* the `i`, `f`, `a` or `L` edit descriptors to indicate how many times they are to be repeated. Thus the format

```
(i5, i5, i5, f6.2, f6.2, f6.2) => (3i5, 3f6.2)
```

has identical meaning.

Read, **write** and **print** statements

A more general for of **Read** statement

read(*cilist*) input_list

where *cilist* is **a control information** list consisting of one or more items, known as **specifiers**, separated by commas. There must always be a unit specifier in the control information list, which takes the form:

unit = *unit*

where *unit* is the input device (or *unit* in F parlance) from which input is to be taken *unit* may also be the name of an **internal file**. It either takes the form of a scalar integer expression whose value is zero or positive, or it may be an asterisk to indicate that the default input unit is to be used. Normally some units will be **preconnected** and will be automatically available to all programs.

The default input will usually be preconnected as unit 1 or unit 5. (This is purely for historical reasons, since IBM, and several other manufacturers, used unit 5 for the card reader and unit 6 for the printer in their early Fortran systems). So it may be written

```
unit = 5
```

or

```
unit = *
```

to identify the default input unit.

Normally the input will need to be converted from some *external* form such as the characters sent by a keyboard, to an *internal* form suitable for storing in the computer's memory. To carry out this conversion it needs a format, and this is identified by a format specifier, which takes one of the forms

```
fmt = ch_expr
```

in an analogous fashion to the format specification discussed earlier in this section. These statements are identical in their effect to the earlier list-directed input statements.

```
read(unit=*, fmt=*) a, b, c  <==>  read *, a, b, c
```

The remaining specifier is concerned with monitoring the outcome of the reading process, and takes the form

`iostat = io_status`

where *`io_status`* is an integer variable. At the conclusion of the execution of the read statement *`io_status`* will set to a value which the program can use to determine whether any errors occurred during the input process. There are four possibilities:

- The variable is set to zero to indicate that no errors occurred.
- The variable is set to a processor-dependent positive value to indicate that an error has occurred.
- The variable is set to a processor-dependent negative value to indicate that a condition known as an end-of-file condition has occurred.
- The variable is set to a processor-dependent negative value to indicate that a condition known as an end-of-record condition has occurred.

For this chapter the `iostat` is simply used to determine whether or not the reading of data was carried out successfully by testing the value of the variable in an `if` or `case` construct:

```
read (unit=*, fmt="(5f6.3)",iostat=ioerror) p, q, r, s, t
if (ioerror /= 0) then   ! ioerror is non-zero
                         ! Print error is non-zero
                         ! and take remedial action
                         ! before exit from procedure
    .
    .
    .
return
```

Exactly the same specifiers are available, as was the case for the `read` statement, although it is impossible to encounter an end-of-file condition or an end-of-record condition during output. The only other difference is the obvious one that an asterisk as a unit identifier refers to the default *output* unit. The *default output unit is* 6 and therefore the following statements are equivalent:

```
write(unit=6, fmt=*) d, e, f
write(unit=*, fmt=*) d, e, f
print *, d, e, f
```

More powerful formats

In this chapter, considerably complex input and output formats will be described. Probably the most important of these concern are multi-record formats, and the repetition of formats.

In the following statements, 12 real numbers into an array `arr`, of size 12, typed 4 to a line are to be read and it could be written

```
read “(4f12.3)”,arr(1:4)
read “(4f12.3)”,arr(5:8)
read “(4f12.3)”,arr(9:12)
```

However the following statements are identical

```
read “(4f12.3)”,arr  <==> read “(4f12.3)”,arr(1:12)
```

After the read statement has used the format to input four real numbers (which are placed in the first four elements of `arr`) it finds that the input list is not yet exhausted, and that another real number is required.

On input, a / causes the rest of the current record to be ignored and the next input item to be the first item of the *next* record. On output, a / terminates the current record and starts a new one.

Thus the statement

```
read “(3f8.2,/,3i6)”,a, b, c, p, q, r
```

will read three real numbers from the first record and three integers from the second.

Multiple consecutive / descriptors cause input records to be skipped or null (blank) records to be output. Thus the statement

```
read “(3f8.2,/,/,3i6)”,a, b, c, p, q, r
```

will cause three real numbers to be read from the *first* record and three integers from the *third*. The second record will be skipped and not read. Because a sequence of / edit descriptors separated by commas is rather ugly it is permitted to precede a / edit descriptor by a repeat count, in the same way as with `a`, `f`, `i` and `l` edit descriptors

Thus an alternative to the previous statement is

```
read “(3f8.2,2/,3i6)”,a, b, c, p, q, r
```

Multiple / descriptors are particularly useful on output which will produce the output shown in the following figure, if a and b have the values 12.25 and 23.50 respectively.

```
real :: a,b
 a = 12.25
 b = 23.50
 write(unit=6,
c          fmt="(t10,a,3/,a,f6.2,a,f6.2,a"//"f7.2,2/,a,f10.3)")
c      "Multi-record example",
c      " The sum of ",a," and",b," is", a+b,
c      " Their product is",a*b
```

```
           Multi-record example


The sum of  12.25 and 23.50 is  35.75

 Their product is   287.875
```