Introduction to
Scientific & Engineering Computing
BIL 102FE (Fortran) Course
for
Week 10

Dr. Ali Can Takinacı

Assistant Professor

in

The Faculty of Naval Architecture and Ocean Engineering

80626

Maslak – Istanbul – Turkey

USING FILES TO PRESERVE DATA

One of the most important aspects of computing is the ability for a program to save the data that it has been using for subsequent use either by itself or by another program.

This involves the output of the data to a file. Files may be written and read sequentially, the information in a file may be written and read in a random order. In either case the file may be stored permanently within the computer system.

This chapter shows how the `read` and `write` statements discussed in the previous chapter can be used to read data from a file and write data to a file, in a sequential manner, and introduces several additional statements which are required when dealing with files. Random access information will not be discussed here.

Files and records

All the programs that were written up to now have been based on the assumption that when the program is run it reads some data from the keyboard, process it, produces some results which are displayed on the screen. Once the program has finished nothing remains within the computer system.

This ignores two very important aspects of the normal computing process.

1. The first is that there are more than a few lines of data it is usually far more appropriate to type the data into a **file**, and then for the program read the data directly from the file.

2. The second aspect that it must be considered occurs when the results produced by one program, or some of the results, are required as data for another program or another run of the same program.

A sequence of records forms a file, of which there are two types – external and internal.

An external file is identifiable sequence of records, which is stored on some external medium (magnetic tape optical disk etc.).

The **file store** of the computer system is used for this purpose. This consists of special input/output units, usually, though not always, based on magnetic disks, etc. Information may be transferred to and from these units by using **`read`** and **`write`** statements in a similar manner to that used for data and results transferred via the default input and output units.

A record refers a sequence of characters such as a line of typing or printed in a line of results.

A record refers to some defined sequence of characters or values.

F programs may read and write three types of records – formatted, unformatted and endfile records.

Formatted, unformatted and endfile records

- The first type of record is called a formatted record, and consists of a sequence of characters selected from those, which can be represented by the processor being used. A formatted record may be:

- `write(unit=10, fmt=format_var) var_1,var_2,var_3`

- Each such output statement creates a new record or several new records if the format used defines multiple records.

A formatted record is formatted so that it can be represented in a form that human beings (or a different type of computer) can understand.

F contains a second type of record, called an unformatted record, which consists of a sequence of values (in a processor-dependent form) and is essentially a copy of some part, or parts, of the memory of the computer.

An unformatted record can only be produced by an unformatted output statement, which is the same as a formatted write statement but without any format specifier:

```
write(unit=9) var_1, var_2, var_3
write(unit=3, iostat=ios) x, y, z
```

As it might be expected that an unformatted record can only be read by an unformatted input statement:

```
read(unit=9) var_4, var_5, var_6
read(unit=3, iostat=io_status) a, b, c
```

```
read(unit=9) var_4, var_5, var_6
read(unit=3, iostat=io_status) a, b, c
```

One important difference between the input/output of formatted and unformatted records is that whereas a formatted input or output statement may read or write more than one record by use of a suitable format, for example

```
write(unit=3, fmt=”(2i8,/,(4f12.4))”) int_1, int_2
```

an unformatted input or output statement will always read or write exactly *one* record. The number of items in the input list of an unformatted `read` statement must therefore be the same as the number in the output list of the unformatted `write` statement that wrote it, or fewer.

There is a third type of record, which is particularly important for files, which are to be accessed sequentially; this is the endfile record, which is a special type of record, which can occur as the last record of a file and is written by a special statement.

e**`ndfile`**`(`*`auxlist`*`)`

where *`auxlist`* consists of a `unit` specifier and optionally, an iostat specifier, where these specifiers are the same as those already introduced is section 9 for use with a `write` statement.

An endfile record has no defined length, but if it is read by an input statement it will cause **end-of-file condition** which can be detected by an `iostat` specifier in a `read` statement. *If it is not specifically detected in this way an error will occur and the program will fail.*

Connecting an external file to a program and disconnecting it

For any information to be transferred between a file and a program the file must be connected to a unit.

In other words, a logical connection, or relationship, must be established between the file and the unit number that will be used in any `read` or `write` statements, which are to use that file.

This connection is initiated by means of an open statement, which takes the form

```
open(open_specifier_list)
```

where *open_specifier_list* is a list of specifiers, as shown in figure below.

```
unit = unit number
file = file_name
status = file_status
form = format_mode
action = allowed_actions
position = file_positon
iostat = ios
```

The `unit` specifier must be present, and takes the same form as in the `read`, `write` and `endfile` statements.

The `status` specifier must also be present, and specifies what is the status of the file before it is opened by the program. It takes the form

```
status = file_status
```

where *`file_status`* is a character expression, which after removing any trailing blanks, is one of `old`, `new`, `replace` or `scratch`.

Note that because `file_status` is a character expression it can be written

```
status = “old”
status = “new”
```

and so on.

If the `file_status` is `old` the file must already exist

If it is `new` then it *must not already exist*.

If `new` is specified then the `open` statement will attempt to create file, and if successful will change its status to `old`, after which any subsequent attempt to open the file as `new` will fail.

If `file_status` is replace, and the file already exists, then it is deleted and an attempt made to create a new file with the same name; if this is successful the status is changed to `old`. If the file does not already exists then the action taken will be the same as if `new` had been specified.

Finally, if `file_status` is scratch then special un-named file is created for use by the program; when the program ceases execution (or when the file is closed) the file will be deleted and will cease to exit. Such a file can therefore be used as a temporary file for the duration of execution only.

As well as specifying the initial status of the file, it must be also specified what types of input/output operations are allowed with the file. The `action` specifier is used for this purpose, and takes the form.

```
action = allowed_actions
```

where *`allowed_actions`* is a character expression, which, after the removal of any trailing blanks, must take one of the three values `read`, `write` or `readwrite`.

If `allowed_actions` is `read` then the file is to be treated as a *read-only* file, and only `read` statements, together with the two file positioning statements **backspace** and **`rewind`** are allowed on this file. **`Write`** and **`endfile`** statements are not allowed, thus this prevents a program from accidentally overwriting information in the file.

If `allowed_actions` is `write` then the file is to be treated as an *output* file, and only **`write`** and **`endfile`** statement, together with the two file positioning statements **backspace** and **`rewind`** are allowed on this file. **`Read`** statements are not allowed.

If `allowed_actions` is `readwrite` then all input/output are allowed for this file.

If the file status is specified as `scratch` then `allowed_actions` must be `readwrite`.

After all, any other value would be meaningless.

The remaining specifiers are all optional, and enable us to specify various requirements regarding the file that is to be opened and to monitor the opening process itself.

The first of these concerns the type of access that is permitted to the file and takes the form

`access =` *`access_type`*

where *`access_type`* is a character expression which after, the removal of any trailing blanks, must take one of the two values `sequential` or `direct`; if no access specifier is provided it is assumed to be `sequential`.

If file is a specifed to be a sequential file, a position specifier must be included to instruct the open statement where the file is to be initially positioned; this takes the form

`position =` *`file_position`*

where *`file_position`* is a character expression which, after the removal of any trailing blanks, must take one of the values `rewind` or `append`.

If the file did not previously exist and *`file_position`* is `rewind` then the file positioned at its initial point. After all, there is nowhere else to position a new file.

If the file does already exist and *`file_position`* is `rewind` then the file positioned at its initial point and a subsequent `read` or `write` statement will either read the first record in the file, or write a new first record as appropriate.

If the file already exists and `file_position` is `append` then the file is positioned immediately before the endfile record, if there is one, or immediately after the last record of the file (at its terminal point) if there is no endfile record.

A subsequent `write` statement will therefore write the next record immediately after the end of existing information in the file; a read statement would lead to either an error or end-of-file condition since the file has no records remaining to be read other than an endfile record, if one exists.

If a file is specified to have direct access then a position specifier is not permitted.

Files normally have a name by which they are known to the computer system, and this name is specified by using a file specifier, which takes the form

```
file = file_name
```

where *`file_name`* is a character expression, which, after the removal of any trailing blanks, takes the form of a file name for the particular computer system.

If this specifier is not present then `status = “scratch”` must be specified.

Thus if the name of the required file is Imagine1, it could be connected the file of that name to program by means of a statement such as

```
open(unit=9, file="Imagine1",
status="old", action="read",
position="rewind")
```

This will connect unit 9 to the specified file.

Thereafter any input or file positioning statements using unit 9 will read from that file, starting with the first record

It will not be permitted to write anything to the file.

Alternatively, it could be read the name of the required file from the keyboard by a program fragment such as the following

```
print *, “Please give the name of the output file”
read “a”, out_file
open(unit=9,file=out_file,status=”old”,action=”write”,
     position=”append”)
```

The `out_file` is a character variable whose length is great to hold the file name. This will allow only output to the specified file, starting immediately after the existing information recorded on the file.

It is not permitted to specify that the status of a named file is `scratch`.

Because of the different ways in which they are written and read, the records in a file must either all be formatted or all be unformatted, and the specifier

`form =` *`format_mode`*

is used to specify which is required.

The character expression *`format_mode`* must take one of the two values `formatted` or `unformatted.`

If it is omitted, then the file is assumed to be formatted

If it is connected for sequential access but unformatted

If it is connected for direct access

The statement

```
open(unit=9,file=datafile,status=”old”,
action=”read”,position=”rewind”)
```

will connect the file `datafile` to unit 9 as a formatted, sequential access file for input only

On the other hand

```
open(unit=7,status=”scratch”,
form=”unformatted”, action=”readwrite”,
position=”rewind”)
```

will create a temporary scratch file and connect it to unit 7 as an unformatted, sequential access file.

The next specifier, `recl`, behaves slightly differently depending upon whether the file is connected for sequential or direct access. The sequential statement takes the form

```
recl = record_length
```

where *`record_length`* is an integer expression, which defines the maximum length that, the records in the file may have. If the file is a formatted file the length is expressed in characters. If it is unformatted file the length is expressed in processor-defined units. In general, this specifier is not required for sequential files, and its main use in this regard is to limit the size of records in a file which will be transferred to some other processor which places a restriction on the size of records in files.

The final specifier, `iostat`, is concerned with recognising when an error occurs during the connection process. For example if the named file does nor exist or is of the wrong type, and operates in the same way as has already been discussed in connection with the `read`, `write` and `endfile` statements.

Non-zero values may be returned in the event of an error during the opening of a file the execution of the program will be terminated unless it is detected by the program:

```
open(unit=13, file=”Problem_File”,
status=”old”,access=”readwrite”,
         position=”rewind”,iostat=ios)
   if (ios /=) 0 then
print *,”error during opening of ‘Problem_File’”
    .
    .
end if
   ! contiue processing
```

Finally it should be noted two important rules:

- If a file is connected to a unit, then it may not also be connected to another unit

- If a file is connected to a unit then another file may not be connected to the same unit.

If a file is first disconnected from a unit then it may be connected to another unit, and another file may be connected to the first unit.

Up to this point it is assumed that once a file has been opened it will remain open for the remainder of the execution of the program. This is frequently what is required, but there are occasions when it is required to disconnect to specify that some specific action is to take place when such disconnection does takes place. As file, which has been connected to a program by means of an open statement can, therefore, be disconnected by means of a `close` statement, which takes the form

`close (`*`close_specifier_list`*`)`

where the possible specifiers are `unit`, `status` and `iostat`.

The `unit` and `iostat` specifiers take the same form as for the `open` statement, while the `status` specifier is used to determine what is to happen to the file when it has been disconnected from the program. It takes the form

```
status = file_status
```

where *`file_status`* is a character expression, which after the removal of any trailing blanks, is either `keep` or `delete`.

If it is `keep` then the file fill will continue to exist after it has been disconnected from the program.

If it is `delete` then the file fill will cease to exist after it has been disconnected.

Only the `unit` specifier is required in a close statement.

If no status specifier is present the file is closed as though `status=”keep”` had been specified unless it was opened with `status=”scratch”`.

If a file is opened with `status=”scratch”` then it will automatically be deleted when it is disconnected from the program.

File positioning statements

There are often situations in which it is required to alter the position in a file without reading or writing any records, and F provides two additional file positioning statements for this purpose. The first of these

```
backspace (auxlist)
```

causes the file to be positioned just before the *preceding* record (that is, enables the program to read the immediately previously read record again).

As with the `endfile` statement *`auxlist`* consists of a `unit` specifier and, optionally, an `iostat` specifier.

The other file positioning statement is

`rewind (`*`auxlist`*`)`

which causes the file to be positioned just before the first record so that a subsequent input statement will start reading or writing the file from the beginning.

Once again, *`auxlist`* consists of a `unit` specifier and,optionally, an `iostat` specifier.

One important point about the positioning of a file concerns the writing of information to a file in a sequential manner. The rule in F is that

writing a record to a sequential file ***destroys*** *all information in the file after that record.*

Thus it is not possible to use `backspace` or `rewind` in order to position a file so that only one particular record can be overwritten, or so that a particular record or records can be read. If it is required to overwrite individual records selectively within a file then the file must be opened for direct access.

A common use of use of `backspace` in conjunction with `endfile` is to add information at the end of a previously written file, as in the following example;

```
 .
 .
!    read up to end-of-file
     do
     read(unit = 8, iostat = ios)dummy
     if (ios < 0) then
     exit   ! negative ios means end-of-file
     end do
!    backspace to before end-of-file record
     backspace (unit=8)
!    now add new information
     write(unit=8, ...) ...
!    terminate file with an end-of-file ready for
     the next time
     endfile(unit=8)
```