- output redirection with append

```
a.exe >> file.out
```

  * tells a.exe to send WRITE(*,*) output to file.out instead of screen.
  * if file.out exists, the new output will be appended to file.out

- These can be mixed

```
a.exe < file.in > file.out
a.exe < file.in >> file.out
```

OPEN/CLOSE statements

- These are an alternative to file i/o redirection
- Offer more flexibility (for example if your inputs are spread over several files or your outputs need to go to different files)
- can have as many files open at one time as you wish (though there are practical limits from an efficiency viewpoint)
- Look at sum of integers program again

```
INTEGER :: rvalue, total

total = 0

OPEN (10, file = 'sumint.in')
OPEN (11, file = 'sumint.out')
loop1: DO
        read(10,*) rvalue
        IF (rvalue < 0) THEN
            EXIT loop1
        ENDIF
        total = total + rvalue
    ENDDO loop1
    WRITE(11,*) 'total = ', total
    CLOSE(10)
    CLOSE(11)
```

Comments

- OPEN (10, file = "sumint.in")

  - 10: * file ID number
    * must be an integer
    * don't use 5 or 6
    * some compilers don't let you use numbers greater than 99.
    * must not be the same as another file that is currently open.

  - "sumint.in": * file name
    * should be in same directory as the program
    * It can be in a different directory if you give the path to the file (not a good idea most of the time).
    * If file doesn't exist, it will be created.

* you can read from multiple files; how this is structured depends on how the reading happens in your program, for example

```
→ OPEN(10, file='file1.in')
      { statements to read
        all data from file1.in
  CLOSE(10)

  OPEN(10, file='file2.in')
      { statements to read
        all data from file2.in
  CLOSE(10)
```

```
→ OPEN(10, file='file1.in')
  OPEN(11, file='file2.in')
     { Read from file1.in &
       file2.in as needed
  CLOSE(10)
  CLOSE(11)
```

* a CLOSE followed by another OPEN of the same file resets the file pointer to the top of the file

```
OPEN(10, file='f1.in')   ← pointer at top of file f1.in
    {
CLOSE(10)
OPEN(10, file='f1.in')   ← pointer back at top of file f1.in
```

* REWIND(10) ← puts file pointer back to top of file 10.

"Triggerless" data input

- The problem with the simple sum program we are using is that it needs a final value of -1 to trigger the end of input data.
- This program doesn't work if we need the sum of positive and negative values.
- one way to handle more general data would be to do

```
loop1: DO
         WRITE (*,*) 'Enter next value'
         READ (*,*) rvalue
         {{{{ operations to do
         WRITE (*,*) 'more data (1=y, 0=n)?'
         READ (*,*) flag  ------> integer.
         IF (flag == 0) THEN
              EXIT loop1
         ENDIF
       ENDDO loop1
```

The IF block → you can have multiple exit conditions

however, this is twice as tedious due to the need to answer the prompt at the end of the loop.

- we would like a way to read in data from a file (simple columns or tables work best for the approach we will use) where the number of values in the file is unknown (but the data layout is).

- The key to this is to use a variation on the READ statement

(assume input redirection is being used. The program using OPEN/CLOSE would be similar)

```
      INTEGER :: rvalue, total

      total = 0
loop1 DO
         READ(*,*, END=100) rvalue
         total = total + rvalue
      ENDDO
100   CONTINUE
```

NO WRITE prompt since I don't plan on using keyboard input

Comments

- 100 is a line number. Any line can be numbered, but this is usually only done in cases where it is needed
- line numbers must be integers and unique.
- They don't have to be sequential, but they usually are.
  - READ(*,*, END=100) rvalue
    This means "read rvalue, but if there is an input error, jump to line # 100."
  - The loop will jump to 100 when the end of the file is reached (EOF error)
  - CONTINUE is a dummy statement. It does nothing.

- Search for maximum value in a list of numbers

```
INTEGER :: a, b, c, d, e, maxval

IF(a > b) THEN
    IF(a>c) THEN
      IF(a>d) THEN
        IF(a>e) THEN
          maxval = a
        ENDIF
      ENDIF
    ENDIF
ENDIF
IF(b>a) THEN
    IF(b>c) THEN
```

{ ⟶ like the sum program, this is not the way to go about the problem. we need a better way.

- Instead of comparing every number to every other number, compare each number to the current largest number found.
- This is an example of a <u>search process</u>. The underlying logic applies to nearly every type of search.
- A search always needs a starting value (initializing the search)

- assume list is double precision

```
INTEGER :: n
Real(Kind = dp) :: rvalue, maxval

n = 0

loop1 : DO
        READ (*,*, END=100) rvalue
        n = n+1
        IF (n == 1) THEN
            maxval = rvalue          <-- initializing search
        ELSE
            IF (rvalue > maxval) THEN
                maxval = rvalue
            ENDIF
        ENDIF
    ENDDO loop1

100 CONTINUE
    WRITE(*,*) 'max value = ', maxval
```