# CS1110 Introduction to Systematic Programming

## Practical Classes in Week 6 -- Visualizing File I/O

**There is only one hand-out for the labs this week.**
*You should spend some time working through this hand-out and completing any earlier hand-outs. The rest of the time in the labs may be spent on tackling the first coursework assignment .*

In this practical we will be taking a closer look at input from files using the procedures `Get` (etc.) from `CS_Int_IO`, `CS_Flt_IO` and `Ada.Text_IO`. At any stage input is taken from the **current input position** in the file. The current input position is indicated by a **file pointer**. When an file is first opened for input, the current input position is set at the start of the file and as input proceeds the file pointer advances through the file so that each character is only input once. With the facilities provided in the library packages `CS_Int_IO`, `CS_Flt_IO` and `Ada.Text_IO`, it is not possible to move the position of the file pointer back towards the start of the file (without closing the file and then re-opening it).

An Ada program `io_view`[1] **simulates** the operation of the input procedures, showing visually how the file pointer advances through the file as input proceeds.

The input file is displayed at the top of the screen in one-dimensional format with end-of-line markers displayed as the pilcrow symbol ¶ and the end-of-file marker displayed as the section sign §. The current position of the file pointer is indicated by a caret symbol ^.

Input commands are entered from a menu by single key-presses:

```
I     Get   (Integer)
F     Get   (Float)
```

The requested I/O operation is simulated and the value that would have been input is displayed. The file pointer caret then moves to its new position in the file. If an invalid input operation is attempted and an Ada run-time exception would have been raised by the input operation, this is indicated by the `io_view` program. However an exception does not cause `io_view` simulation to crash whereas a real Ada program would terminate in these circumstances.

For convenience there is also an `io_view` command for undoing the previous input command and backing up the file pointer to its previous position. This is invoked with the single key stroke:

```
U     Undo
```

This is useful if an I/O operation performed by mistake (in an `io_view` simulation unlike a real Ada program we can recover from our mistakes!). In fact any number of input operations can be 'undone' by repeatedly pressing the `U` key.

The input file position can be reset to the start of the file with single key stroke:

```
R     Reset
```

and the quit command has the expected effect:

```
Q     Quit
```

Other input procedures for inputting single characters, character strings and whole lines of text can also be emulated by pressing other keys as indicated in the `io_view` menu. Don't worry about these for the moment -- they will be discussed later in the ISP course. For now just concentrate on using `Get` for inputting integers and floats.

**Exercise** Copy the files

```
/usr/local/staffstore/cs1110/tutorial-programs/petrol.dat
/usr/local/staffstore/cs1110/io_view
```

[1] Originally written by Dale Stanborough from University of Perth, Western Australia and adapted for use at Aston by Alan Barnes.

to your own user area. Now change directory (using `cd`) to the directory to which you have just copied the above two files and start the `io_view` program as follows:

```
io_view  petrol.dat
```

The file `petrol.dat` is a data file suitable for input into the petrol price program (Question 1 on Problem Sheet 3) discussed in last week's ISP tutorials. The file contains pairs of `Float` values terminated by a single `0.0` value meant to act as a data terminator.

Try pressing the `F` key several times to simulate the calls to `Get` from `CS_Flt_IO` and note how the file pointer advances through the file.

Now try pressing the `I` key twice to simulate the calls to `Get` from `CS_Int_IO`. Note how the first `Get` succeeds in inputting an `Integer` value but leaves the input position immediately before a decimal point. The second `Get` raises the exception `Data_Error` to indicate that the data in the file is not suitable for the attempted operation -- in this case we have data of the form `.52` which is not a valid integer value. In real life our Ada program would have crashed at this point.

Undo the erroneous operation by pressing the `U` key. Now press the `F` key and note that a value such as `.52` is interpreted as the float value `0.52`.

Now advance the file pointer to just before the terminating `0.0` value by pressing the `F` key repeatedly.

Then try the effect of pressing the `F` key twice more. The first operation inputs `0.0` as we might expect, but the second one raises the exception `End_Error` indicating that we have attempted to read a non-existent data value and have 'run off' the end of the file. Again a real Ada program would have crashed at this point.

**Using `io_view` as an Algorithm Design/Debugging Aid**

We can use `io_view` to help with the design/debugging of the input steps in an algorithm. The advantage is that we can run through and test the input steps of an outline algorithm whereas with `gdbtk` we need a complete program to test[2]. For example we can trace through outline algorithms such as:

**Algorithm 1**

```
Volume : Float;
PricePaid : Float;
................
Get(Item => Volume);
Get(Item => PricePaid);
WHILE Volume /= 0.0 LOOP
   Process this data pair
   Get(Item => Volume);
   Get(Item => PricePaid);
END LOOP;
```

**Algorithm 2**

```
Volume : Float;
PricePaid : Float;
................
Get(Item => Volume);

WHILE Volume /= 0.0 LOOP
   Get(Item => PricePaid);
   Process this data pair
   Get(Item => Volume);
END LOOP;
```

using `io_view` and discover that the first algorithm will crash at the end of the data file `petrol.dat` whereas the second one behaves correctly.

Similarly if we mistakenly define `Volume` and/or `PricePaid` to be of type `Integer` then `io_view` will reveal that the I/O operations do not behave correctly even in algorithm 2. If both variables are defined to be of type `Integer` then `Data_Error` is raised by the second `Get` (i.e. the second press of `I`). If one variable is defined to of type `Integer` and the other of type `Float` then alternate presses of `I` and `F` will soon reveal that the I/O operations get 'out of sync' with the structure of the data.

The program `io_view` can also simulate the effect on the file pointer of input of single characters, of strings, and of whole lines. These features of Ada will be covered later in the module in units 14 and 16. **To save disk space delete `io_view` at the end of the practical.**

[2] Of course io_view can only detect logic errors in the input steps. It cannot detect errors in pure data processing steps; for this we of course need to use gdb.

Next in this practical we use `io_view` to study the effect of the input procedures

> `Get` for character input
> `Skip_Line` for discarding the rest of the current input line

and the predicate functions `End_Of_Line` and `End_of_File` for testing whether the current input position is at the end of a line or at the end of the file. It may help to refer to Unit 14 as you work through this exercise.

1. Copy the file `/usr/local/staffstore/cs1110/test1.dat` to your own user area and display the file in normal "two-dimensional" form in a terminal window using a utility such as `cat`, `less` or `more`.

2. Open another terminal window and resize it so that it is a reasonable size (around half the screen size[3]) and then start `io_view` (remember to `cd` to the directory in which `io_view` resides)

```
        io_view test1.dat
```

and now arrange the two terminal windows so that the contants of both are visible.

`io_view` displays a one-dimensional representation of the file `test1.dat` at the top of the screen with end-of-line markers displayed as the pilcrow symbol ¶ and end-of-file displayed as the section sign §.

3. Note how <u>empty lines</u> (such as line 3) in the normal two-dimensional display of the file appear as consecutive ¶ symbols in the one-dimensional display, whereas <u>blank lines</u> (lines containing only blank characters such as line 5) appear as two ¶ symbols separated by blanks. Blank and empty lines, of course, look the same on the normal 2-D display.

4. Simulate the effect of `Get` for `Character` input in the `io_view` by pressing the `C` key several times and note how the current input position (displayed as the caret symbol `^`) advances through the file one character at a time and note how blanks are treated as "full value" characters by this version of `Get`.

On the other hand, `Get` for `Integer` input (`I` key) or `Float` input (`F` key) skip over blanks and input complete numbers and so (usually) "consume" more than one character.

Recall that in an `io_view` simulation (unlike an Ada program), you can undo the effect of the last input operation by pressing the `U` key (`Undo`) and can return to the start of the file by pressing the `R` key (`Reset`).

5. Note how when the file pointer reaches the end of a line (that is reaches a ¶ symbol) that `End_of_Line` becomes `True` whereas at other positions on the line, `End_of_Line` is `False`.

6. Note that when the current input position is at the end of a line, `Get` for `Character`, `Integer` or `Float` all skip to the start of the next line and input the next data value from there.

Any empty lines will be skipped over by any `Get` operation and furthermore `Get` for `Integer` or `Float` will also skip over any blank lines (whereas `Get` for `Character` inputs the blanks one by one).

7. Simulate the effect of `Skip_Line` by pressing the `K` key and note how the input position moves to the start of the next line (that is the input position advances to just past the next end of line marker ¶).

8. Advance the input position using `Skip_Line` so that it is positioned at the start of the last line in the file (that the line which contains `51 One Two).` Now press the `I` key twice (`Get` for `Integer`) and note that a `Data_Error` is raised as `One` is not a valid `Integer.` However as one would expect using `Get` for `Character` (`C` key) can handle this data.

[3] If the window is too small, `io_view`'s screen update does not work correctly.

9. Advance the input position using `Get` for `Character` so that it is positioned before the last end-of-line marker in the file. Note how `End_of_Line` is `True` and `End_of_File` is `True`. A further `Get` operation at this point will cause the `End_Error` exception to be raised.

10. Quit `io_view` by pressing the `Q` key.

## 2nd Coursework Problem

Try running io_view with the small data file lotto.dat provided for test purposes with the second coursework assignment. You will need a number of "Get Character" calls to process the date (how many?) then you will need a number of "Get Integer" calls to input the 6 main ball numbers and the bonus ball. Note there is no problem with the spaces between these integer values as "Get Integer" simply skips leading blanks when inputting an integer. After the seven ball numbers have been input End_Of_Line will be true on a normal week but false if it is a roll-over week (as there will still be characters on the current line which have yet to be input). A call to Skip_Line will advance to the start of the next line (whether it is a normal or roll-over week) so that we are ready to input the next date.

## Badly-formed Input Files

As indicated at the end of unit 14 trailing blanks at the end of a line and trailing empty or blank lines at the end of a file are "bad news" and can cause problems for data processing as they make it difficult to detect the End_Of_Line or End_Of_File in loops.

Generally data files should not contain any trailing blanks nor trailing empty or blank lines to avoid such problems. The following exercise illustrates some of these problems

Copy the file /usr/local/staffstore/cs1110/test2.dat to your own user area and display the file in normal "two-dimensional" form in a terminal window and in "one-dimensional" form in io_view. This is essentially the same file as test1.dat, but some lines (lines 1 and 2) containing trailing blank characters (that is blanks after the last number on the line) and the file ends with several blank and empty lines. Note how the "two-dimensional" display of the file appears visually the same as the first file. However the file appears differently in the "one-dimensional" format.

1. Advance the input position to just after the last number on line 1 and note how End_of_Line is False (as there are blanks remaining on the line after the input position). A Get for Integer or Float will skip to the next line and input the next number from there. Thus a loop of the form:

```
WHILE NOT End_Of_Line LOOP
   Get(Number);
   Process this Number
END LOOP;
```

will not behave correctly in the presence of trailing blanks on a line as it advances past the end of the current line. There is no problem for character input and a loop such as

```
WHILE NOT End_Of_Line LOOP
   Get(Char);
   Process this Char
END LOOP;
```

will behave correctly in the presence of trailing blanks on a line.

2. Advance the input position so that it is positioned at the end of the line (which contains One Two) and note how End_of_File is False as there are some empty lines after this line.

A Get for Character, Integer or Float will cause an End_Error to be raised as the empty lines will be skipped as Get "looks for" the next (non-existent) data value in the file. Thus a loop of the form:

```
WHILE NOT End_Of_File LOOP
   Get(Data);
   Process this Data
END LOOP;
```

will not behave correctly in the presence of trailing empty lines in the file.