AQA
Realising potential

Teaching guide: Input/output file handling

This resource will help with understanding input/output and file handling. It supports Section 3.2.7 of our current GCSE Computer Science specification (8520). The guide is designed to address the following learning aims:

- Review how the keyboard and the screen are the standard input and output devices.
- Recognise the advantages of using persistent data storage with programs.
- Read data from, and write data to, a text file.

Input/output (I/O)

Computer programs take inputs, process data and produce outputs. The inputs to a program can originate from many locations: a microphone, a mouse, a motion sensor, a heat sensor, an input stream over a network, a digital camera, another file and so on. However, the 'standard' form of input is from a keyboard. Likewise, data can be output to many devices: speakers, printers, another file, actuators and so on but the 'standard' output is the monitor (or screen). There is a later section on input and output devices that cover these in more detail.

We have used two commands that enable data to come in or go out of a program:

```
USERINPUT
OUTPUT
```

Because the monitor and keyboard are considered standard input and output respectively, it is assumed that the data for `USERINPUT` comes from the keyboard and the data for `OUTPUT` is displayed on a monitor. Strictly speaking, standard output is normally the text terminal but this is invariably displayed on the monitor.

Persistent data storage

Every program that we have written so far in these training guides performs actions and processes data and then stops. At the point the program stops all of the data that has been processed, including all of the values stored in variables and data structures, will be lost.   With the majority of the programs and applications that are used on a daily basis it is clear that some data needs to be stored after the programs have finished executing; persistence is the term used to mean storing data between instances of a running program.

Using persistent data storage with programs provides obvious advantages: games can remember what level you left on or what the highest scores are, music-playing applications can remember your favourite genre or artist, web browsers can remember your log in details to a frequently accessed website and so on. There are many ways that data can be stored and retrieved by programs – many languages have their own serialization methods that store a representation of the actual variable or data structure itself (for instance the use of ‘pickle’ in Python). This section will explore how data can be saved to, and read from, a text file. A text file does not need to contain recognisable words – the term is more general and loosely means any file whose contents is intended to be read as characters as opposed to files that just contain 1s and 0s (binary files) that have no direct character representation.

File input/output is so programming language specific that it is impossible to be specific without contradicting the way that different languages handle reading and writing to text files. Fortunately, all of the languages supported by the specification have extensive documentation on file handling. Having said this, there are some general principles that are worth remembering:

1. Files are addressed by their name such as iosection.txt. If your computer program is run from the same folder or directory as this file then just the filename will generally suffice. However, if the program and the text file are located in different folders on your computer or network then you will have to provide more details in the name (to be absolutely unambiguous on your computer system you could provide the fully qualified filename such as /users/aqa/trb/chapter1/iosection.txt but bear in mind that if you change the location of your file then this fully qualified filename will be incorrect).
2. If you write the instructions in your program to access a file and that file does not exist (at least not in the location you specified) then this will probably cause a run-time error and your program will most likely crash. In the next section we will look at ways to validate data including checking a file’s existence prior to opening a connection to it.
3. If you open a connection to a file for reading and writing it is good practice to close the connection within your program as soon as the connection is no longer needed. Some languages will do this automatically, others may require you to close the connection explicitly.
4. Be aware that if more than one program or user is trying to access the same file at the same time then complications can arise.
5. Data is sometimes written to, and read from, files as a string – this may mean that you need to convert the datatype of your input and output in order to use them as intended in your program.

The following example shows how writing a player name and their high score (with a comma between them to indicate where the name ends and the high score begins) to a file called `scores.txt` could take place.

```
# assume that the string variable name contains the player name and
# the integer variable score contains their high score
file ← OPEN('scores.txt')
data_to_write ← name + ',' + INT_TO_STRING(score)
WRITE(file, data_to_write)
CLOSE(file)
```

Reading the data from this file and storing the two items of data in variables of the correct type could look like this (we'll assume that the file only contains one player name and their high score):

```
file ← OPEN('scores.txt')
data_in ← READLINE(file)
CLOSE(file)
pos_comma ← POSITION(data_in, ',')
player_name ← SUBSTRING(data_in, 0, pos_comma-1)
high_score_as_string ← SUBSTRING(data_in, pos_comma+1, LEN(data_in)-
1)
high_score ← STRING_TO_INT(high_score_as_string)
```

Study your own language's documentation carefully as it will almost certainly differ from these examples. It will also probably contain methods that simplify how to read and write data in common formats such as when values in a text file are separated from each other using commas, as used in the previous example (comma-separated values or CSV) or using other common formats such as JSON.

In the example above there is only one line of data stored in the text file – of course it is more normal to have many lines of text that represent different records within the same file and your programs will have to deal with this. Reading from a file can normally be done line-by-line or in one go (commonly `READLINE` and `READ` or `READALL` respectively). Reading line-by-line commonly goes within a loop that terminates when the entire file is read; as always, check your language's documentation to explore this