A practical introduction to Small Basic, focusing on manipulating data in arrays and introducing the development of a graphical user interface.

Preparation required:
Small Basic available on all computers, all sample programs available in a shared repository.
Shuffle Array Template and cards for pairs / groups.
One Hundred Doors Problem per person.

Linear Data Structures

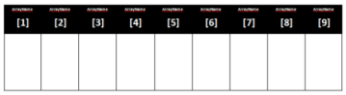
So far, our conceptual model for programming has highlighted the 'Big 3' constructs for creating algorithms (sequence, selection and repetition), and the notion of a variable. We introduced primitive data types pointing out that the commonly used string variable is really a combination of characters. It is a compound data structure. There are many different compound data structures supported by different languages. We introduce a linear structure, which Small Basic calls an array. To keep things simple we can think of this as a list of values. Imagine we wanted to store a list of 200 student names, or a set of 500 scores. Rather than creating an individual variable for each student, or each score, we can use arrays. Each array has a name, and the individual items are referenced by their position in it – known as the array index. In Small Basic, the index positions start at 1, much like lists in Scratch. Many languages use array indexing starting at 0.

An array is a simple concept; a collection of related data which avoids the problem of managing many individual variable names. Nonetheless many children will have difficulty manipulating the data in them correctly. Following Seymour Papert, the best way for children to internalise the concept is through a variety of playful practical activities.

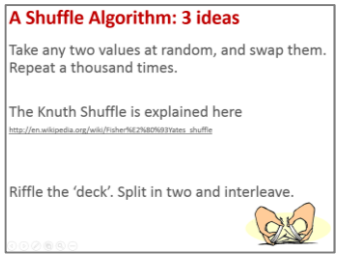
Investigating Shuffles

We start by considering a simple subroutine called Many_Integers. This can be found in the zipped folder 0301Investigating_Shuffles_Programs. The subroutine is defined at the top and called in the main program. It is best to read the code as a group and predict its behaviour before running it. It prompts the user to input the number of items required, then creates an array with integers incrementing from 1 to the number required. It is a useful technique for filling an array with a series of integers. The key point is to highlight the use of the LoopCounter to identify the index position in the array.

The shuffling activity is substantial and brings together many of the elements we have highlighted in both parts of this Unit. A shuffle algorithm can be developed with varying levels of complexity. There are many ways to shuffle cards, or other lists of items. This is a good group activity to encourage thinking about the stages involved in a shuffle. There are no right answers, but the challenge is to articulate their method as an algorithm. Using just 9 values initially, we can use integers 1 through 9 to represent them in an array. There is a printed array that can be used with playing cards with values 1-9. This will help articulate the steps involved – or dry run their algorithm (or that of their neighbours) to test it. If they are stuck for ideas, a slide gives three possible shuffle algorithms.



The first suggestion is the easiest to implement, requiring just a series of random swaps.

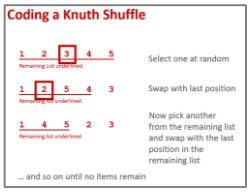


Implementing the Knuth, or Fisher Yates shuffle is more challenging. A series of slides are included for you to use or study.

Finally, there are some interesting properties associated with riffle shuffles, which we will explore in a final exercise. Implementing a riffle shuffle is a good exercise for more able children.

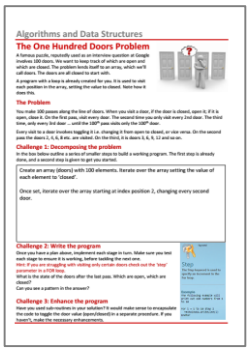
Before we attempt to code a shuffle subroutine, ensure the problem of swapping values in an array is understood; a third, temporary variable is required to avoid the second value being overwritten. By first defining a swap subroutine, we can keep the shuffle code clear of clutter and focus solely on the manipulation of the array. (There is an ingenious way of swapping two numeric values without using a third variable, which is revealed in the slide notes.)

The practical challenge is to code a working shuffle. We can use an array of 52 integers to simulate the 52 different cards in a deck. Do we need to code this from scratch? Students should recognise that they can use their ManyInteger subroutine in this new program. Example code for a simple shuffle is also included in the program resources. Slides are included as a walk-through of the Knuth shuffle. Coding a Knuth shuffle is a good exercise for more able students and for staff learning about array manipulation. The key to understanding this shuffle is identifying the remaining list. It is this that makes it an improvement on a random swap. If this is to be used as an exercise, and the students are struggling, identifying the variables required can help. The final code is included on a slide for reference. Note the use of Step to control how you iterate over the array. This is important in our next challenge.



Manipulating arrays isn't easy. Children need repeated exposure to simple examples. Shuffling allows for solutions of varying complexity. It is a lot easier to implement than sorting. Sorting algorithms often rely on the use of nested constructs which can be a step too far for many children.

One Hundred Doors



This activity poses a classic problem. To solve it involves nested constructs. It is a difficult challenge, aimed at teachers, to illustrate the extra cognitive load involved, but may be appropriate for able students. An activity sheet is included. Reputedly used as an interview question at Google, it involves 100 doors. We want to keep track of which are open and which are closed. The problem lends itself to an array, which we'll call doors. The state of each door is initially shut. A counter controlled loop can be used to reference each index position, setting the value to closed.

Here's the problem: You make 100 passes by the doors. On the first pass, visit every door. If the door is closed, open it; if open, close it. The second time you only visit every 2nd door. The third time, only every 3rd door … until the 100th pass visits only the 100th door. Every visit to a door involves toggling it i.e. changing it from open to closed, or vice versa. On the second pass the doors 2, 4, 6, 8 etc. are visited. On the third, it is doors 3, 6, 9, 12 etc. What is the state of the doors after the last pass? Which are open, which are closed?

Discuss how we might approach this. We clearly need to decompose the problem. The discussion should make it clear that this is a difficult challenge. It is aimed at teachers, but may be appropriate for able students. A good solution would involve a nested loop. The activity sheet could be used if you wish to set this as an independent challenge.

The code is also included; DoorsSolution.sb. If time is limited, the solution can be discussed, rather than setting as a practical activity. It could be treated as a reading task, or as a dry-run so people appreciate how it works. There are 2 key elements for understanding the program. Line 16 uses the incremented value of the start variable to control the step. Lines 17 to 21 toggle the door value.

We could we make it simpler to understand using procedural abstraction. A refined version defines a subroutine (Small Basic's term for a named block of code) for toggling the value of a door. Notice how, by abstracting away the detail, it makes the main code easier to understand.

Once working, check whether anyone can spot the pattern in the final configuration of doors. The only doors open are square numbers.