```
    /* assume that A[0] to A[n-1] have valid values */
    didswaps = 1;
    while (didswaps) {
        didswaps = 0;
        for (i=0; i<n-1; i++) {
            if (A[i] > A[i+1]) {
                int_swap(&A[i], &A[i+1]);
                didswaps = 1;
            }
        }
    }
```

```
vice: ./bubblesort
Enter as many as 10 values, ^D to end
1 8 15 3 17 12 4 8 4
^D
9 values read into array
Before:   1  8 15  3 17 12  4  8  4
After :   1  3  4  4  8  8 12 15 17
```

Figure 7.3: Program fragment showing how to sort an array from smallest to largest using the bubble sort algorithm. Function `int_swap` is defined in Figure 6.8 on page 95. The lower box shows an execution of a program containing the fragment.

were out of order, then the array is sorted. In this latter case, at the end of the inner `for` loop, variable `didswaps` will still retain the zero (false) value with which it was initialized at the start of the loop, and the outer `while` loop can terminate. Note the use of the function `int_swap` from Figure 6.8 on page 95 – there is no point not making ongoing use of functions that were written for other initial purposes. Good software design facilitates this kind of re-use.

The name "bubble sort" comes about because of the way that the largest remaining item is bubbled toward the top of the array at each iteration of the `while` loop. This is the spiraling convergence referred to in Chapter 4 – each time the `while` loop repeats, one more item in the array reaches its final resting position. Hence, it is possible to guarantee that the array is sorted after `n-1` iterations of the `while` loop.

As an example, suppose that the seven numbers $\{22, 14, 17, 42, 27, 28, 23\}$ are to be sorted. Table 7.1 illustrates the passes that are required. Items shaded in grey move one spot to their left in that pass as larger items step over them to the right. A total of four passes are required: three to get the array into sorted order, and then a fourth to establish that no more swaps are required.

The discussion in this chapter is more about arrays than about sorting, and while the exercises at the end of this chapter introduce other sorting algorithms, they too are of the "agricultural" quality of bubble sort rather than being "formula one" performers. Sorting algorithms that can be used to efficiently sort very large sets of values – the Ferraris of sorting – are discussed in detail in Chapter 12. Those clever algorithms are to bubble sort what binary search is to linear search, and mean that sorting is not usually a bottleneck operation in data processing applications.

	`A[0]`	`A[1]`	`A[2]`	`A[3]`	`A[4]`	`A[5]`	`A[6]`
Initially	22	14	17	42	27	28	23
After pass 1	**14**	**17**	22	**27**	**28**	**23**	42
After pass 2	14	17	22	27	**23**	28	42
After pass 3	14	17	22	**23**	27	28	42
After pass 4	14	17	22	23	27	28	42

Table 7.1: Tracing the action of bubble sort on the array $\{22, 14, 17, 42, 27, 28, 23\}$. Items in grey swap one location to their left during that pass. The fourth pass finds that no further swaps are required.

To understand why bubble sort is considered to be slow, look again at Table 7.1. Small objects move only one position to their left during each pass. Hence, if the smallest object is initially in the rightmost position, then $n - 1$ passes are required to sort an array of n objects. Moreover, each pass involves $n - 1$ comparisons. To sort $n = 100$ objects takes as many as $99^2 \approx 9{,}800$ comparisons, and completes in far less than a second of computation time (recall from Chapter 1 that a rule of thumb for current computers is around 10 million steps per second). But to sort $n = 100{,}000$ items using bubble sort will take rather longer: as many as $999{,}999^2 \approx 10^{10}$ comparisons, or perhaps $1{,}000$ seconds. The key point to note is that the bound on the execution cost grows quadratically in the number of objects in the array. This point is returned to in Chapter 12 when more efficient sorting algorithms are introduced.

Bubble sort is one of many different sorting algorithms. It has the advantage of being simple to understand and to implement, but is slow to execute if more than a few thousand objects are being sorted.

7.4 Arrays and functions

This section discusses one of the genuinely insightful decisions that was made during the development of the C language. It concerns the relationship between arrays and pointers, and the way arrays are handled in functions.

Suppose that `A` is an array of `n` integer values. The first variable in `A` is `A[0]`, and is stored at location `&A[0]`. Similarly, the second variable is `A[1]`, stored at `&A[1]`. The developers of C specified that the array itself, `A`, is defined to be a *pointer constant* whose value is the address of the first variable in the array, that is, `&A[0]`. As an example, if `p` is a variable of type pointer to `int` (that is, `p` is of type `int*`), then the assignment `p=A` has exactly the same effect as `p=&A[0]` – it leaves `p` pointing at the first variable in `A`.

The identifier used as an array name is a constant of type pointer to T, where T is the type underlying the array.

This relationship makes it easy to pass arrays into functions. Since the array name is a pointer constant, if the array is passed into a function, what is transferred into the