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 corresponding argument variable is a pointer expression. It can either be declared as a pointer in the function header, or as an undimensioned array using the notation `A[]`. If the argument is declared as a pointer, and then altered within the function (by being assigned to), it is only the local variable that is changed – the original address constant is not altered in any way. In this sense, array arguments are treated exactly the same as integer arguments – the value of the expression is copied over into a local argument variable that can then be altered, but the original expression cannot be altered.

Array argument variables can be declared as either arrays, or as pointers of the same underlying type. The number of elements in the array does not need to be part of the declaration of that argument, and the function can accept array arguments of any size, provided the underlying type is the same.

Functions that receive the argument as a pointer variable are shown later in the chapter, and in the remainder of this section the functions shown declare the argument to be an undimensioned array. For example, consider the program in Figure 7.4. That program is the one presumed to have been executed in the lower box of Figure 7.3, and makes use of a total of four functions. Because an array is a pointer to the first of its elements, changes made in the function via that pointer are directly reflected in the underlying array. That means that use of an array within a function follows an identical pattern to use of the array in the scope in which it was declared, and the bodies of functions `read_int_array` and `sort_int_array` are exactly as discussed already in Figures 7.2 and 7.3. The third function, `print_int_array`, similarly uses the argument variable `A` to access elements of the array declared in the `main` function, in this case to print them out.

Access from within a function to array elements via a pointer passed as an argument directly alters the underlying variables in the original array.

Note the manner in which the reading function is instructed of the maximum number of values that the array can hold (10 in the example execution shown in

```
/* Read an array, sort it, write it out again.
*/
#include <stdio.h>

#define MAXVALS 10

int read_int_array(int A[], int n);
void sort_int_array(int A[], int n);
void print_int_array(int A[], int n);
void int_swap(int *p1, int *p2);

int
main(int argc, char *argv[]) {
    int numbers[MAXVALS], nnumbs;
    nnumbs = read_int_array(numbers, MAXVALS);
    printf("Before: ");
    print_int_array(numbers, nnumbs);
    sort_int_array(numbers, nnumbs);
    printf("After : ");
    print_int_array(numbers, nnumbs);
    return 0;
}

int
read_int_array(int A[], int maxvals) {
    int n, excess, next;
    /* the body of this function is in Figure 7.2 */
    return n;
}

void
sort_int_array(int A[], int n) {
    int i, j;
    /* the body of this function is in Figure 7.3 */
}

void
print_int_array(int A[], int n) {
    int i;
    for (i=0; i<n; i++) {
        printf("%4d", A[i]);
    }
    printf("\n");
}
```

**Figure 7.4:** A complete program for sorting a set of integers. The body of function `read_int_array` is shown in Figure 7.2, and the body of function `sort_int_array` is shown in Figure 7.3. The lower box of Figure 7.3 shows an execution of the completed program.