```
/* Show use of the functions fopen, fwrite, and fread.
*/
#include <stdio.h>
#include <assert.h>

#define SIZE 7
#define FILENAME "temp.dat"

void print_doubles(double*, int);

int
main(int argc, char **argv) {
    double A[SIZE];
    FILE *fp;
    int i;
    /* initialize the array with some values */
    for (i=0; i<SIZE; i++) {
        A[i] = 1.2345*i + 0.6789;
    }
    print_doubles(A, SIZE);
    /* open the file for writing */
    fp = fopen(FILENAME, "w");
    assert(fp != NULL);
    /* write the whole array in one operation */
    i = fwrite(A, sizeof(*A), SIZE, fp);
    assert(i == SIZE);
    /* clear the array */
    for (i=0; i<SIZE; i++) {
        A[i] = 0.0;
    }
    print_doubles(A, SIZE);
    /* open the file for  reading */
    fp = freopen(FILENAME, "r", fp);
    assert(fp != NULL);
    /* read the array  back*/
    i = fread(A, sizeof(*A), SIZE, fp);
    assert(i == SIZE);
    print_doubles(A, SIZE);
    fclose(fp);
    return 0;
}
```

```
  0.679  1.913  3.148  4.382  5.617  6.851  8.086
  0.000  0.000  0.000  0.000  0.000  0.000  0.000
  0.679  1.913  3.148  4.382  5.617  6.851  8.086
```

Figure 11.3: Using `fopen`, `fwrite`, `freopen`, and `fread` to manipulate a binary file. The lower box shows an execution of the program in the upper box. Function `print_doubles` uses a `printf` inside a loop to write the array contents as text to `stdout`.

When working with binary files in which all of the objects are the same type, the option of both reading from and writing to the same file is possible – using the file a little like an array is used in memory. To allow *random access*, all of the `"r"`, `"w"`, and `"a"` mode options can be modified by the addition of a "+", to make `"r+"`, `"w+"`, and `"a+"`. This flag indicates a file that will be used for both input and output. For example a file opened as `"w+"` is truncated to zero bytes, if it already exists, and any `fprintf` or `fwrite` operations extend the file from there. But `fscanf` and `fread` operations can also be performed on the file, intermixed with the output operations. The only requirement is that an `fseek` operation be performed when there is a change from input to output operations, or vice versa, so that the next record to be processed is unambiguously identified.

Function `fseek` shifts the current location in a stream. The first argument is a stream pointer, and the second an integer offset to specify the size of the move. The third argument is one of three symbolic constants defined in `stdio.h`: `SEEK_SET`, which indicates positive locations relative to the start of the file; `SEEK_CUR`, which indicates positive and negative locations relative to the current location in the file; or `SEEK_END`, which indicates positive or negative locations relative to the current end of the file. For example, the call `fseek(fp,-sizeof(staff_t),SEEK_CUR)` shifts the current location back by exactly the size of one `staff_t` structure. So if a `staff_t` record had just been read using `fread`, it can be modified in memory, and then written back to the file, overwriting the old record that was in its place. Another `fseek` should then be executed prior to the next `fread` operation.

> Files can be opened for interleaved input and output operations. Function `fseek` is used to move the current location within the file. This mode of operation usually only makes sense if all objects in the file are of the same type.

11.3 Case study: Merging multiple files

Now for a more complex example involving files. Consider this next specification:

> Write a program that reads a set of text files, with the lines in each file presumed to be in sorted order; and writes a single sorted file to the standard output. Lines may be assumed to be at most 1024 characters long. As many as ten input files must be allowed for, specified as command-line arguments.

Figure 11.4 shows an example of the desired behavior. In the example, four text files `f1.txt`, `f2.txt`, `f3.txt`, `f4.txt` are each in sorted order. The program `mergefiles` is then executed, with those four filenames passed as command-line arguments. The output of `mergefiles`, written to `stdout`, is the sorted combination of all of the lines in the four input files. In this application the sorting is based purely upon the alphabetic characters in each line, so that `"dolphins eat"` precedes `"dolphins live"`. A more sophisticated merging program would allow the user to specify where in each file the sort key was located, and would also offer the option