```
void print_planet(planet_t one_planet);
planet_t read_planet(void);
int read_planet_ptr(planet_t *one_planet);

int
main(int argc, char *argv[]) {
    planet_t planet;
    planet = read_planet();
    print_planet(planet);
    if (read_planet_ptr(&planet) != EOF) {
        print_planet(planet);
    }
    return 0;
}

void
print_planet(planet_t one_planet) {
    /* the body of this function is in Figure 8.3 */
}

planet_t
read_planet(void) {
    planet_t new_planet;
    /* the body of this function is in Figure 8.4 */
    return new_planet;
}

int
read_planet_ptr(planet_t *planet) {
    int nvals_read;
    printf("Enter %s:\n", PLANETPROMPT);
    nvals_read = scanf("%s %s %lf %lf %lf",
        planet->name,
        planet->orbits,
        &planet->distance,
        &planet->mass,
        &planet->radius);
    if (nvals_read != 5) {
        return EOF;
    } else {
        return 0;
    }
}
```

Figure 8.6: Program showing use of structure arguments in functions. Function `read_planet` returns a structure; whereas function `read_planet_ptr` is passed a pointer to a structure, and uses that pointer to fill in the fields of the structure. The structure declaration appears in Figure 8.2. The body of function `print_planet` appears in Figure 8.3, and the body of function `read_planet` appears in Figure 8.4.

meaning that requiring that a pointer to a variable (rather than a general expression) be present in the argument list is not likely to be restrictive in any way.

For most purposes, it is appropriate to pass a structure pointer rather than a structure. Passing a pointer allows the function to alter the components in the underlying compound variable.

8.4 Structures and arrays

There is one last, but very important, way in which structures can be used – they can be replicated in arrays. Consider again the example `typedef` shown in Figure 8.2 on page 130. A program manipulating planetary data is unlikely to be dealing with the earth alone, or even the sun, earth and moon – if it were, it would hardly be worth the trouble of declaring the structure type. Much more likely is that information about the 20 or 100 most massive objects in the solar system is being maintained, perhaps in a program that calculates the times of solar eclipses, or the gravitational forces acting on the space shuttle in orbit. An array is the perfect solution:

```
#define MAXBODIES 100

int nplanets=0;
planet_t planets[MAXBODIES];
```

As always with arrays, a variable is used to count the current number of things stored in the array (integer `nplanets`); and the same care to avoid overflow must be taken when reading in to an array of structures as is taken when reading into an array of simpler variables (Section 7.2 on page 101).

When an array of structures is declared, it becomes even more natural to use structure pointers. For example, the function call `read_planet_ptr(planets+i)` passes a pointer to the `i`'th element of `planets`, and reads values into the five variables that make up `planets[i]`.

The array subscripting operator has the same precedence as the component selection operator, and both are left associative. That means that parentheses to force ordering are usually unnecessary – expressions involving a mix of the "`[]`" subscripting, and "`.`" and "`->`" selection operators, can be read from left to right. For example, `planets[0].orbits` is the `orbits` component of the first object in array `planets`; and `planets[0].orbits[3]` is the fourth `char` variable in that string.

Arrays of structures are an important data organization technique. They represent many real-world situations in which large quantities of data are to be handled.

As with any array, an array of structures needs to be accompanied by a variable that indicates how many entries in the array are currently valid, declared as `nplanets` in the example above. But the two objects `nplanets` and `planets` belong together. So it makes sense for them to be joined in a structure, and a single aggregate type created that includes a description of its own size: