In Figure 6.3, variable `z` is declared prior to any of the functions, and given an initial value of 2. That initialization happens even before the `main` begins executing. So the first `printf` executed shows `z` with the value 2. The calls to the function alter that global variable, and the modified value is available for use in `main` after the function returns. On the other hand, variable `x` in `func` is still a local variable, and its changed value is always lost when the function returns.

> Global variables can be accessed and modified from within any function defined in the same file as the variable.

All of the functions declared in a file are also global. They can be called from within any other function in the same file, and can also be called (using separate compilation) by functions declared in other files.

The ability to declare global variables does raise one problem: it is possible to have multiple variables declared with the same name. To resolve this ambiguity, the rules of scope stipulate that any reference to a multiply declared variable applies to the local one; and a global variable can only be referred to if there is no local variable of the same name (irrespective of type) that *shadows* it. The same rule applies to function names, and can be a cause for confusion – declaring a variable called `sqrt` in a function automatically makes the `sqrt` routine from the mathematics library unavailable in that function. If you do declare a variable called `sqrt`, and then later call the function `sqrt`, the compiler objects at that point with a message something like "called object is not a function", because, in the scope in question, `sqrt` is not a function and hence cannot be called.

> Local variable declarations in a function shadow global variables and functions of the same name, and render them inaccessible to the function.

More subtle errors arise from the unexpected shadowing of global variables, or even of argument variables, when there is no such type clash identified by the compiler. For example, it is legal in C to declare a local variable that has the same name as an argument variable in the same function, in which case the argument variable is shadowed. A good C compiler will issue a warning message in this situation (`gcc -Wall` certainly does); nevertheless, you should remain alert to the possibility that you may have inadvertently shadowed a global or argument variable.

Does this mean that if we want to write functions that have side effects, we must make use of global variables? Fortunately, the answer is "no". Sections 6.6 and 6.7 describe a rather more elegant way of allowing a function to alter the environment it got called from, and that is the mechanism that should be used when side effects are required.

> Global variables should be used only with extreme caution. Relying on them for communications to and from a function makes the function less general, less robust, and harder to reuse.

It is also worth repeating some advice that was given earlier: always read the warning messages issued by the compiler. Warning-free code may be hard to achieve

in some circumstances, but for the programs being written in this book, should be possible. And even if you are unable for some reason to eliminate all of the warnings, you must at least make sure you understand why they are arising.

> Pay attention to compiler warning messages – they indicate things that might go wrong.

6.5 Static variables

If a global variable is being used exclusively by one function, as a kind of "internal" memory, then having it accessible to other functions is unnecessary. To handle this requirement, C provides a third *storage class*, and allows the declaration of `static` variables. A static variable is local to the function it is declared in, but retains its value between calls to that function. Because it is to retain its value, it is not allocated in the stack frame for the function, and is instead given space within a separate area of memory reserved for static and global variables. Figure 6.4 shows a simple program that makes use of a static variable. As with the other example programs in this chapter, you are encouraged to trace through the function calls to ensure that you understand why the various `printf` statements produce the values that they do.

```
int
main(int argc, char *argv[]) {
    int x=3, z=5;
    printf("main: x=%2d, z=%2d\n", x, z);
    func(x);
    printf("main: x=%2d, z=%2d\n", x, z);
    func(z);
    printf("main: x=%2d, z=%2d\n", x, z);
    return 0;
}

void
func(int x) {
    static int z=7;
    x = x+1;
    z = x+z+1;
    printf("func: x=%2d, z=%2d\n", x, z);
}
```

```
main: x= 3, z= 5
func: x= 4, z=12
main: x= 3, z= 5
func: x= 6, z=19
main: x= 3, z= 5
```

Figure 6.4: A program fragment showing the declaration and use of a static variable. The lower box shows an execution of the program.