```
int z=2;

int
main(int argc, char **argv) {
    int x=3;
    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) {
    x = x+1;
    z = x+z+1;
    printf("func: x=%2d, z=%2d\n", x, z);
}
```

```
main: x= 3, z= 2
func: x= 4, z= 7
main: x= 3, z= 7
func: x= 8, z=16
main: x= 3, z=16
```

Figure 6.3: A program fragment showing that all functions in a file have access to global variables. The lower box shows an execution of the program.

The ability to declare global variables does raise one problem: it is possible to have multiple variables declared with the same name. Figure 6.4 shows an example program that does this. In Figure 6.4, variable `z` is global. But the function also declares a variable `z`, which is local. 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 (perhaps because it is to hold the square root of some value being manipulated in the function) automatically makes the `sqrt` routine from the mathematics library unavailable in that function. You need to choose variable names with a little bit of care:

```
    double sqrt, x=10.0;
    sqrt = M_PI + sqrt(x);
```

This program fragment results in an error message from the C compiler:

```
sqrtvar.c: In function ‘main’:
sqrtvar.c:10: called object is not a function
```

The error arises because, in this scope, `sqrt` is not a function and cannot be called.

```
int z=2;

int
main(int argc, char **argv) {
    int x=3;
    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) {
    int z=7;
    x = x+1;
    z = x+z+1;
    printf("func: x=%2d, z=%2d\n", x, z);
}
```

```
main: x= 3, z= 2
func: x= 4, z=12
main: x= 3, z= 2
func: x= 3, z=11
main: x= 3, z= 2
```

Figure 6.4: A program fragment showing the shadowing of a global variable. The lower box shows an execution of the program.

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 sparingly. Relying on them for communications to and from a function makes the function less general, less robust, and harder to reuse.