The `static` Storage Class

The `static` storage class can be used with all variables, regardless of where they're declared, but it has a different effect on a variable declared outside a block than it does on a variable declared inside a block. When used *outside* a block, the word `static` specifies that a variable has internal linkage. When used *inside* a block, `static` changes the variable's storage duration from automatic to static. The following figure shows the effect of declaring `i` and `j` to be `static`:

```
static int i;      <- static storage duration, file scope, internal linkage

void f(void)
{
  static int j;    <- static storage duration, block scope, no linkage
}
```

When used in a declaration outside a block, `static` essentially hides a variable within the file in which it's declared: only functions that appear in the same file can see the variable. In the following example, the functions `f1` and `f2` both have access to `i`, but functions in other files don't:

```
static int i;

void f1(void)
{
  /* has access to i */
}

void f2(void)
{
  /* has access to i */
}
```

This use of `static` can help implement a technique known as information hiding.

information hiding ➤ 19.2

A `static` variable declared within a block resides at the same storage location throughout program execution. Unlike automatic variables, which lose their values each time the program leaves the enclosing block, a `static` variable will retain its value indefinitely. `static` variables have some interesting properties:

- A `static` variable in a block is initialized only once, prior to program execution. An `auto` variable is initialized every time it comes into existence (provided, of course, that it has an initializer).
- Each time a function is called recursively, it gets a new set of `auto` variables. If it has a `static` variable, on the other hand, that variable is shared by all calls of the function.