- Although a function shouldn't return a pointer to an `auto` variable, there's nothing wrong with it returning a pointer to a `static` variable.

Declaring one of its variables to be `static` allows a function to retain information between calls in a "hidden" area that the rest of the program can't access. More often, however, we'll use `static` to make programs more efficient. Consider the following function:

```
char digit_to_hex_char(int digit)
{
  const char hex_chars[16] = "0123456789ABCDEF";

  return hex_chars[digit];
}
```

Each time the `digit_to_hex_char` function is called, the characters `0123456789ABCDEF` will be copied into the `hex_chars` array to initialize it. Now, let's make the array `static`:

```
char digit_to_hex_char(int digit)
{
  static const char hex_chars[16] = "0123456789ABCDEF";

  return hex_chars[digit];
}
```

Since `static` variables are initialized only once, we've improved the speed of `digit_to_hex_char`.

The `extern` Storage Class

The `extern` storage class enables several source files to share the same variable. Section 15.2 covered the essentials of using `extern`, so I won't devote much space to it here. Recall that the declaration

```
extern int i;
```

informs the compiler that `i` is an `int` variable, but doesn't cause it to allocate memory for `i`. In C terminology, this declaration is not a *definition* of `i`; it merely informs the compiler that we need access to a variable that's defined elsewhere (perhaps later in the same file, or—more often—in another file). A variable can have many *declarations* in a program but should have only one *definition*.

There's one exception to the rule that an `extern` declaration of a variable isn't a definition. An `extern` declaration that initializes a variable serves as a definition of the variable. For example, the declaration

```
extern int i = 0;
```

is effectively the same as

```
int i = 0;
```