Mastering C declarations takes time and practice. The only good news is that there are certain things that can't be declared in C. Functions can't return arrays:

```
int f(int)[];        /*** WRONG ***/
```

Functions can't return functions:

```
int g(int)(int);     /*** WRONG ***/
```

Arrays of functions aren't possible, either:

```
int a[10](int);      /*** WRONG ***/
```

In each case, we can use pointers to get the desired effect. A function can't return an array, but it can return a *pointer* to an array. A function can't return a function, but it can return a *pointer* to a function. Arrays of functions aren't allowed, but an array may contain *pointers* to functions. (Section 17.7 has an example of such an array.)

Using Type Definitions to Simplify Declarations

Some programmers use type definitions to help simplify complex declarations. Consider the declaration of x that we examined earlier in this section:

```
int *(*x[10])(void);
```

To make x's type easier to understand, we could use the following series of type definitions:

```
typedef int *Fcn(void);
typedef Fcn *Fcn_ptr;
typedef Fcn_ptr Fcn_ptr_array[10];
Fcn_ptr_array x;
```

If we read these lines in reverse order, we see that x has type `Fcn_ptr_array`, a `Fcn_ptr_array` is an array of `Fcn_ptr` values, a `Fcn_ptr` is a pointer to type `Fcn`, and a `Fcn` is a function that has no arguments and returns a pointer to an `int` value.

18.5 Initializers

For convenience, C allows us to specify initial values for variables as we're declaring them. To initialize a variable, we write the = symbol after its declarator, then follow that with an initializer. (Don't confuse the = symbol in a declaration with the assignment operator; initialization isn't the same as assignment.)

We've seen various kinds of initializers in previous chapters. The initializer for a simple variable is an expression of the same type as the variable:

```
int i = 5 / 2;    /* i is initially 2 */
```