Keep in mind that gifts itself is an array, not a structure, so constructions such as gifts.price are not valid.

To initialize an array of structures, combine the rule for initializing arrays (a brace-enclosed, comma-separated list of values for each element) with the rule for structures (a brace-enclosed, comma-separated list of values for each member). Because each element of the array is a structure, its value is represented by a structure initialization. Thus, you wind up with a brace-enclosed, comma-separated list of values, each of which itself is a brace-enclosed, comma-separated list of values:

```
inflatable guests[2] = // initializing an array of structs
{
    {"Bambi", 0.5, 21.99}, // first structure in array
    {"Godzilla", 2000, 565.99} // next structure in array
};
```

As usual, you can format this the way you like. Both initializations can be on the same line, or each separate structure member initialization can get a line of its own, for example.

**Bit Fields**

C++, like C, enables you to specify structure members that occupy a particular number of bits. This can be handy for creating a data structure that corresponds, say, to a register on some hardware device. The field type should be an integral or enumeration type (enumerations are discussed later in this chapter), and a colon followed by a number indicates the actual number of bits to be used. You can use unnamed fields to provide spacing. Each member is termed a *bit field*. Here's an example:

```
struct torgle_register
{
    int SN : 4;       // 4 bits for SN value
    int : 4;          // 4 bits unused
    bool goodIn : 1;  // valid input (1 bit)
    bool goodTorgle : 1; // successful torgling
};
```

You use standard structure notation to access bit fields: