Although it's convenient to be able to use an enumeration value as an integer, it's dangerous to use an integer as an enumeration value. For example, we might accidentally store the number 4—which doesn't correspond to any suit—into `s`.

Using Enumerations to Declare "Tag Fields"

Enumerations are perfect for solving a problem that we encountered in Section 16.4: determining which member of a union was the last to be assigned a value. In the `Number` structure, for example, we can make the `kind` member an enumeration instead of an `int`:

```
typedef struct {
  enum {INT_KIND, DOUBLE_KIND} kind;
  union {
    int i;
    double d;
  } u;
} Number;
```

The new structure is used in exactly the same way as the old one. The advantages are that we've done away with the `INT_KIND` and `DOUBLE_KIND` macros (they're now enumeration constants), and we've clarified the meaning of `kind`—it's now obvious that `kind` has only two possible values: `INT_KIND` and `DOUBLE_KIND`.

Q & A

Q: **When I tried using the `sizeof` operator to determine the number of bytes in a structure, I got a number that was larger than the sizes of the members added together. How can this be?**

A: Let's look at an example:

```
struct {
  char a;
  int b;
} s;
```

If `char` values occupy one byte and `int` values occupy four bytes, how large is `s`? The obvious answer—five bytes—may not be the correct one. Some computers require that the address of certain data items be a multiple of some number of bytes (typically two, four, or eight, depending on the item's type). To satisfy this requirement, a compiler will "align" the members of a structure by leaving "holes" (unused bytes) between adjacent members. If we assume that data items must