the arguments to the correct type, if possible.

We've already discussed how to handle correctly the return value. Let's look now at what happens when you use the wrong number of arguments. For example, suppose you made the following call:

```c
double z = cube();
```

Without function prototyping, the compiler lets this go by. When the function is called, it looks where the call to `cube()` should have placed a number and uses whatever value happens to be there. This, for example, is how C worked before ANSI C borrowed prototyping from C++. Because prototyping is optional for ANSI C, this still is how some C programs work. But in C++ prototyping is not optional, so you are guaranteed protection from that sort of error.

Next, suppose you provide an argument but it is the wrong type. In C, this could create weird errors. For example, if a function expects a type `int` value (assume that's 16 bits) and you pass a `double` (assume that's 64 bits), the function looks at just the first 16 bits of the 64 and tries to interpret them as an `int` value. C++, however, automatically converts the value you pass to the type specified in the prototype, provided that both are arithmetic types. For example, Listing 7.2 manages to get two type mismatches in one statement:

```c
cheers(cube(2));
```

First, the program passes the `int` value of 2 to `cube()`, which expects type `double`. The compiler, noting that the `cube()` prototype specifies a type `double` argument, converts 2 to 2.0, a type `double` value. Then, `cube()` returns a type `double` value (8.0) to be used as an argument to `cheers()`. Again, the compiler checks the prototypes and notes that `cheers()` requires an `int`. It converts the return value to the integer 8. In general, prototyping produces automatic type casts to the expected types. (Function overloading, discussed in Chapter 8, "Adventures in Functions," can create ambiguous situations, however, that prevent some automatic type casts.)

Automatic type conversion doesn't head off all possible errors. For example, if you pass a value of 8.33E27 to a function that expects an `int`, such a large value cannot be converted correctly to a mere `int`. Some compilers warn you of possible data loss when there is an automatic conversion from a larger type to a smaller.

Also, prototyping results in type conversion only when it makes sense. It won't, for