example, convert an integer to a structure or pointer.

Prototyping takes place during compile time and is termed *static type checking*. Static type checking, as we've just seen, catches many errors that are much more difficult to catch during runtime.

## Function Arguments and Passing by Value

It's time to take a closer look at function arguments. C++ normally passes arguments *by value*. That means the numeric value of the argument is passed to the function, where it is assigned to a new variable. For example, Listing 7.2 has this function call:

```cpp
double volume = cube(side);
```

Here `side` is a variable that, in the sample run, had the value 5. The function heading for `cube()`, recall, was this:

```cpp
double cube(double x)
```

When this function is called, it creates a new type `double` variable called `x` and assigns the value 5 to it. This insulates data in `main()` from actions that take place in `cube()`, for `cube()` works with a copy of `side` rather than with the original data. You'll see an example of this protection soon. A variable that's used to receive passed values is called a *formal argument* or *formal parameter*. The value passed to the function is called the *actual argument* or *actual parameter*. To simplify matters a bit, the C++ standard uses the word *argument* by itself to denote an actual argument or parameter and the word *parameter* by itself to denote a formal argument or parameter. Using this terminology, argument passing assigns the argument to the parameter. (See Figure 7.2.)

*Figure 7.2. Passing by value.*