Thus, `local()` displays `warming` as 0.8, but it displays `::warming` as 0.4. You'll encounter this operator again in namespaces and classes.

**Global or Local?**

Now that you have a choice of using global or local variables, which should you use? At first, global variables have a seductive appeal—because all functions have access to a global variable, you don't have to bother passing arguments. But this easy access has a heavy price—unreliable programs. Computing experience has shown that the better job your program does of isolating data from unnecessary access, the better job the program does in preserving the integrity of the data. Most often, you should use local variables and pass data to functions on a need-to-know basis rather than make data available indiscriminately with global variables. As you will see, OOP takes this data isolation a step further.

Global variables do have their uses, however. For example, you might have a block of data that's to be used by several functions, such as an array of month names or the atomic weights of the elements. The external storage class is particularly suited to representing constant data, for then you can use the keyword `const` to protect the data from change.

```c
const char * const months[12] =
{
   "January", "February", "March", "April", "May",
   "June", "July", "August", "September", "October",
   "November", "December"
};
```

The first `const` protects the strings from change, and the second `const` makes sure that each pointer in the array remains pointing to the same string to which it pointed initially.