When a function returns a reference or a pointer to a data object, that object had better continue to exist once the function terminates. The simplest way to do that is to have the function return a reference or pointer that was passed to it as an argument. That way, the reference or pointer already refers to something in the calling program. The use() function in Listing 8.6 uses this technique.

A second method is to use new to create new storage. You've already seen examples in which new creates space for a string and the function returns a pointer to that space. Here's how you could do something similar with a reference:

```c++
sysop & clone(sysop & sysopref)
{
    sysop * psysop = new sysop;
    *psysop = sysopref;    // copy info
    return *psysop;        // return reference to copy
}
```

The first statement creates a nameless sysop structure. The pointer psysop points to the structure, so *psysop is the structure. The code appears to return the structure, but the function declaration indicates the function really returns a reference to this structure. You then could use the function this way:

```c++
sysop & jolly = clone(looper);
```

This makes jolly a reference to the new structure. There is a problem with this approach, which is that you should use delete to free memory allocated by new when the memory is no longer needed. A call to clone() hides the call to new, making it simpler to forget to use delete later. The auto_ptr template discussed in Chapter 16, "The String Class and the Standard Template Library," can help automate the deletion process.

What you want to avoid is code along these lines:

```c++
sysop & clone2(sysop & sysopref)
{
    sysop newguy;           // first step to big error
    newguy = sysopref;      // copy info
    return newguy;          // return reference to copy
}
```