however, the * symbol performs indirection (when used as a unary operator). The statement

```
*p = &i;   /*** WRONG ***/
```

would be wrong, because it assigns the address of i to the object that p points to, not to p itself.

Q: Is there some way to print the address of a variable? [p. 244]

A: Any pointer, including the address of a variable, can be displayed by calling the printf function and using %p as the conversion specification. See Section 22.3 for details.

Q: The following declaration is confusing:

```
void f(const int *p);
```

Does this say that f can't modify p? [p. 251]

A: No. It says that f can't change the integer that p *points to*; it doesn't prevent f from changing p itself.

```
void f(const int *p)
{
  int j;

  *p = 0;    /*** WRONG ***/
  p = &j;    /* legal */
}
```

Since arguments are passed by value, assigning p a new value—by making it point somewhere else—won't have any effect outside the function.

***Q: When declaring a parameter of a pointer type, is it legal to put the word const in front of the parameter's name, as in the following example?**

```
void f(int * const p);
```

A: Yes, although the effect isn't the same as if const precedes p's type. We saw in Section 11.4 that putting const *before* p's type protects the object that p points to. Putting const *after* p's type protects p itself:

```
void f(int * const p)
{
  int j;

  *p = 0;    /* legal */
  p = &j;    /*** WRONG ***/
}
```

This feature isn't used very often. Since p is merely a copy of another pointer (the argument when the function is called), there's rarely any reason to protect it.

An even greater rarity is the need to protect both p *and* the object it points to, which can be done by putting const both before and after p's type: