in a program, we'd probably expect `f` to change the value of `x`. It's possible, though, that `f` merely needs to examine the value of `x`, not change it. The reason for the pointer might be efficiency: passing the value of a variable can waste time and space if the variable requires a large amount of storage. (Section 12.3 covers this point in more detail.)

We can use the word `const` to document that a function won't change an object whose address is passed to the function. `const` goes in the parameter's declaration, just before the specification of its type:

Q&A

```
void f(const int *p)
{
  *p = 0;    /*** WRONG ***/
}
```

This use of `const` indicates that `p` is a pointer to a "constant integer." Attempting to modify `*p` is an error that the compiler will detect.

11.5 Pointers as Return Values

We can not only pass pointers to functions but also write functions that *return* pointers. Such functions are relatively common; we'll encounter several in Chapter 13.

The following function, when given pointers to two integers, returns a pointer to whichever integer is larger:

```
int *max(int *a, int *b)
{
  if (*a > *b)
    return a;
  else
    return b;
}
```

When we call `max`, we'll pass pointers to two `int` variables and store the result in a pointer variable:

```
int *p, i, j;
...
p = max(&i, &j);
```

During the call of `max`, `*a` is an alias for `i`, while `*b` is an alias for `j`. If `i` has a larger value than `j`, `max` returns the address of `i`; otherwise, it returns the address of `j`. After the call, `p` points to either `i` or `j`.

Although the `max` function returns one of the pointers passed to it as an argument, that's not the only possibility. A function could also return a pointer to an external variable or to a local variable that's been declared `static`.