As it's now written, the result of the division—an integer—will be converted to `float` form before being stored in `quotient`. We probably want `dividend` and `divisor` converted to `float` *before* the division, though, so that we get a more exact answer. A cast expression will do the trick:

```
quotient = (float) dividend / divisor;
```

`divisor` doesn't need a cast, since casting `dividend` to `float` forces the compiler to convert `divisor` to `float` also.

Incidentally, C regards (*type-name*) as a unary operator. Unary operators have higher precedence than binary operators, so the compiler interprets

```
(float) dividend / divisor
```

as

```
((float) dividend) / divisor
```

If you find this confusing, note that there are other ways to accomplish the same effect:

```
quotient = dividend / (float) divisor;
```

or

```
quotient = (float) dividend / (float) divisor;
```

Casts are sometimes necessary to avoid overflow. Consider the following example:

```
long i;
int j = 1000;

i = j * j;    /* overflow may occur */
```

At first glance, this statement looks fine. The value of `j * j` is 1,000,000, and `i` is a `long`, so it can easily store values of this size, right? The problem is that when two `int` values are multiplied, the result will have `int` type. But `j * j` is too large to represent as an `int` on some machines, causing an overflow. Fortunately, using a cast avoids the problem:

```
i = (long) j * j;
```

Since the cast operator takes precedence over `*`, the first `j` is converted to `long` type, forcing the second `j` to be converted as well. Note that the statement

```
i = (long) (j * j);    /*** WRONG ***/
```

wouldn't work, since the overflow would already have occurred by the time of the cast.