the output is:

```
x=12300000000000000.0000000
y=12300000000000004.0000000
z=                4.0000000
```

and it is clear that it is not possible to add small values to large numbers and expect accurate answers. You need to remember that fact when you are planning programs. Floating point arithmetic is risky, especially if the large values in question are later reduced to small values by further operations. Accumulated rounding errors have the potential to make any answer that you compute meaningless.

Variables of type `double` store approximate values over a much larger range than do `int` variables.

If reduced numeric precision can be tolerated, C offers a third numeric data type: the `float`. Variables of type `float` require less memory in the computer than do those of type `double`, but unless they are being used in very large arrays (Chapter 7), the difference in cost is small, and it makes more sense to declare `double`s.

Constants also have types. All of the following are numeric constants:

```
    1                               -22
    345                             0
    3.14159                         10000.
    -.12345                         1e10
    -2.72e+0                        1.0e-6
    156e+52                         -2.78e-17
```

The "`e`" that appears in some of the numbers is an *exponent part*, and represents a scale factor as a power of ten. For example, `2.4e5` represents the number $2.4 \times 10^5 =$ 240,000, and `1e-6` represents $1.0 \times 10^{-6} = 0.000001$ (and not 1^{-6}).

Any numeric value that contains neither a decimal point nor an exponent part is inferred to be of type `int`. Numeric values that contain either a decimal point or an exponent are of type `double`. It is also possible to explicitly declare numeric constants in several other ways, including as octal and hexadecimal integers. These options are beyond the scope of this book.

Use of inappropriate constants can create problems. For example, to implement the physics calculation $e = \frac{1}{2}mv^2$ to work out the kinetic energy e of a mass m moving at velocity v, it is tempting to write:

```
    double energy, mass, velocity;
    /* BEWARE -- incorrect code */
    energy = (1/2)*mass*velocity*velocity;
```

The problem is that in this form both `1` and `2` are integers, so the first operation carried out is an integer division, and an initial result of integer zero obtained. Only when the multiplication by `mass` is considered is the integer value in the first subexpression converted to a `double`. By then it is too late – zero as an `int` converts to zero as a `double`, and the damage has been done. The fix is simple once the mistake is noticed – the factor `(1/2)` can be replaced by `0.5`, for example. But until the mistake is identified, the program is erroneous. Notice also that the C compiler is completely silent about the potential problem here. You have to be aware of this kind of thing as you write the program. The next section considers expressions, and type conversions during the evaluation of expressions, in more detail.

Be sure that you use constants of types that match the variables they are being combined with.

Character constants can also be created. Single quotes are used to delineate them: `'a'`, `'8'`, `' '`, and so on. Because the backslash character and quote character have special meanings, they are *escaped* to make character constants: `'\\'` and `'\''` respectively. Other special characters are newline, `'\n'`; and tab, `'\t'`. The corresponding variables are declared as `char`, and are read with a "`%c`" format descriptor.

2.3 Operators and expressions

The programs you have already seen in this chapter show some of the numeric operations possible in C. Table 2.1 lists all of the arithmetic operators, and gives examples of their use.

The multiplicative operators "`*`", "`/`", and "`%`", have a higher *precedence* than the additive operators "+" and "−". Operators of equal precedence are evaluated in left-to-right order. For example, `2+3*4` evaluates to `14`; and `15-6-2` evaluates to `7`; and `15%4*16%9` evaluates to `3`. Evaluation order can always be adjusted through the insertion of parentheses: `(2+3)*4` is `20`, and `(15%4)*(16%9)` is `21`.

As already noted, division when both operands are of type `int` results in an integer quotient being calculated and then being used in the next part of the computation: `1/2*2` is `0`, and `1/2*2.0` is `0.0`. The next code fragment shows another example:

```
int n_pass, class_size;
double pass_percent;
/* BEWARE -- incorrect code */
pass_percent = n_pass/class_size*100;
```

One solution is to force, via the rules themselves, the desired result:

```
pass_percent = 100.0*n_pass/class_size;
```

Another is to use an explicit *cast* operation, which forces a type conversion:

```
pass_percent = (double)n_pass/class_size*100;
```

Casting is a *unary* (one operand) operation in the same way that the "−" in `x=-y` is a unary operator. Any type can be used in a cast, with some conversions being more sensible than others.

Suppose instead that an integral pass percentage is required, rounded off to the nearest integer. Even more care is required:

```
int n_pass, class_size, pass_percent;
pass_percent = (int)(0.5 + 100.0*n_pass/class_size);
```