Worth commenting on is the unfortunate reuse of an operator symbol for a quite different purpose: "&" also means "address of", and so the compiler must determine which interpretation to apply to an "&" operator by examining the context in which it is used. This issue is referred to as *overloading*, and can cause confusion. Similarly, in C the operator "*" means both multiplication and address dereferencing.

Left shift operations are equivalent to multiplying by a power of two, and the similar right shift operator ">>" has the effect of dividing by a power of two. There are also bitwise operators for the "or" of two integers, "|"; and for the bitwise complement of an integer, "~". For example, 1101 | 1001 yields 1101, and ~ 1101 is 0010. Finally, there is a bitwise *exclusive or* operator "^" that sets a bit in the result to 1 if the corresponding bits in the two operands differ: 1101 ^ 1001 gives 0100.

The use of "&" as shown in Figure 13.1 is sometimes known as a *masking* operation, since it selects from the first operand the bits that match places where the second operand has a 1 bit. Using these two bit operations, the program in Figure 13.1 extracts in turn each bit of the integer value `val`, and prints either a zero or a one.

Looking at the output, some of the earlier overflow behavior of integers can now be explained. The $w = 32$ binary equivalent for thirteen is easily checked, and, with a bit more work, the value for one million can also be verified. The last three values represent some of the limiting values for 32-bit integers, and fit the general pattern shown in Table 13.3. Can you see now why the program fragment on page 16 gives the results it does?

By now you will also have realized that the C type `unsigned` declares an unsigned integer variable. The right shift operator ">>" has different behavior on signed values than on unsigned values, and you need to be careful – any variables that are participating in these logical operations should be declared to be `unsigned`. Using the $w = 32$ bit wordsize associated with most computer hardware, an `unsigned` variable can take on values that are between 0 and $2^{32} - 1 = 4{,}294{,}967{,}295$. It is also possible to declare `unsigned char` variables, and `long` and `short` integers. None of the byte sizes for these different integer types are tightly specified, but a `char` variable will usually have $w = 8$ bits; a `short` variable is often represented in $w = 16$ bits; and `int` and `long` integers in $w = 32$ bits. Some C systems also offer a `long long` type which might provide $w = 64$ bits. All of these integer types can also be `unsigned`. For example, an `unsigned char` variable takes only positive values, and on most hardware can hold a number between 0 and 255.

Variables that are declared as `unsigned` types store positive values only.

The two floating point types `float` and `double` are represented in three parts: a *sign*, an integer *exponent*, and a fractional *mantissa*. Typically in a `float` one bit is used for the sign; seven bits are used for the exponent, which is either a power of two or a power of 16, and stored as an integer; and another 24 bits for the mantissa, stored as a value between zero and one either as 24 bits following an implicit binary point, or as 6 hexadecimal (base 16) digits after an implicit radix-16 point. Twenty-four binary digits corresponds to approximately seven decimal digits of accuracy. In a `double`, the sizes of the fields allocated to the exponent and mantissa are approximately doubled into a 64 bit two-word quantity, and the number of decimal digits of

Number (decimal)	Number (binary)	Exponent (decimal)	Mantissa (binary)	Representation (bits)
0.5	0.1	0	.100000000000	0 000 1000 0000 0000
0.375	0.011	-1	.110000000000	0 111 1100 0000 0000
3.1415	11.001001000011$\cdots$	2	.110010010000	0 010 1100 1001 0000
-0.1	-0.0001100110011$\cdots$	-3	.110011001100	1 101 1100 1100 1100

Table 13.4: Floating point representation when $w_s = 1$, $w_e = 3$ and $w_m = 12$, and the exponent is a binary numbers stored using w_e-bit twos-complement representation, and the mantissa is a w_m-bit binary fraction.

precision that can be manipulated before rounding errors intrude also approximately doubles.

Variables of type `float` and `double` are stored as an integer exponent part, and a fractional mantissa part.

As an example, suppose that on some computer `float` variables are stored in 16 bits, with $w_s = 1$ bit used for a sign, a $w_e = 3$ bit binary exponent part, and a $w_m = 12$ bit fractional binary mantissa. Table 13.4 shows the bit representation for four values. Note how even a number as simple as one tenth in decimal has a non-terminating binary representation. The last column shows the 16-bit combination that would actually be stored in this hypothetical representation. The inverse value for the last entry is

$$-1 \times 2^{-3} \times (2^{-1} + 2^{-2} + 2^{-5} + 2^{-6} + 2^{-9} + 2^{-10})$$

which in decimal is calculated as

$$-0.125 \times (0.5 + 0.25 + 0.03125 + 0.015625 + 0.001953125 + 0.0009765625)$$

and equals $.0999755859375$. Can you see now where floating point rounding errors come from?

To complete this section, Table 13.5 lists all of the C operators, and their precedence, and replaces the earlier version of the same table given on page 31. The same advice as has already been given continues to apply: if in doubt, parenthesize.

13.3 The C preprocessor

The `#define` was introduced in Chapter 2. It is one of the facilities provided by the *preprocessor*, the first pass of the C compiler. The preprocessor searches for lines that start with a "#", and removes them from the file that is passed onto the main part of the compiler. Those lines are instead taken to be directives that determine the final form of what gets compiled. For example, the value established by a `#define` directive is replaced everywhere that identifier appears in the program, and provides symbolic constants.