float and double

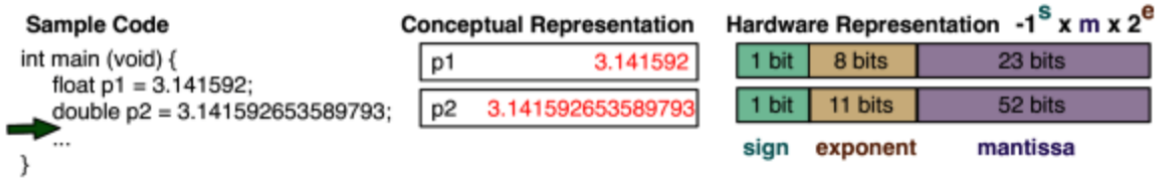
The final two basic data types in C allow the programmer to express real numbers. Since there are an infinite number of real numbers, the computer cannot express them all (that would require an infinite number of bits!). Instead, for values that cannot be represented exactly, an approximation of the value is stored.

If you think about the fact that computers can only store values as 0s and 1s, you may wonder how it is possible to store a real number, which has a fractional part. In much the same way that decimal representations of a number can have a fractional portion with places to the right of a decimal point (the tenth's, hundredth's, thousandth's, etc. places), binary representations of numbers can have fractional portions after the binary point. The places to the right of the binary point are the half's, quarter's, eighth's, etc. places.

One way we could (but often do not) choose to represent real numbers is fixed point. We could take 32 bits, and interpret them as having the binary point in the middle. That is, the most significant 16 bits would be the "integer" part, and the least 16 bits would be the "fractional" part. While this representation would be conceptually simple, it is also rather constrained—we could not represent very large numbers, nor could we represent very small numbers precisely.

Instead, the most common choice is similar to scientific notion. Recall that in decimal scientific notation, number 403 can be expressed as 4.03\times 10^{2}. Computers use floating point notation, the same notation but implicitly in base 2: m\times 2^{e}. m is called the mantissa (though you may also hear it referred to as the significand). e is the exponent.

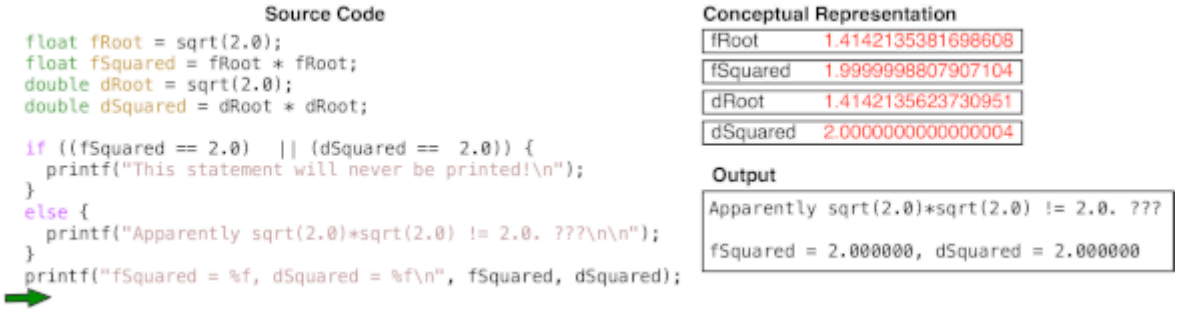
A float has 32 bits used to represent a floating point number. These 32 bits are divided into three fields. The lowest 23 bits encode the mantissa; the next 8 bits encode the exponent. The most significant bit is the sign bit, s, which augments our formula as follows: (-1)^{s}\times m\times 2^{e}. (When s = 1, the number is negative. When s = 0, the number is positive.) A double has 64 bits and uses them by extending the mantissa to 52 bits and the exponent to 11 bits. Examples of both a float and a double are shown in the figure below.



Standards. There would be many possible ways to divide a given number of bits into the mantissa and exponent fields. The arrangement here is part of the IEEE (Institute of Electrical and Electronics Engineers) Standard. Industry standards like these make it

possible for engineers from a variety of companies to agree upon a single encoding by which floating point numbers can be represented and subsequently interpreted across all languages, platforms, and hardware products. Part of the IEEE Standard for floating point notation involves two adjustments to the bit-wise representations of a float and a double. These adjustments (normalization and adding a bias) make the actual binary representation of these numbers less accessible to a first time observer. We encourage the interested reader to read the actual IEEE floating point Standard and allow the less curious reader simply to trust that there is a bit-wise encoding for the numbers in the figure above, which is just outside the scope of this course.

Precision. There are an infinite number of values between the numbers 0 and 1. It should be unsurprising, then, that when we use a finite number of bits to represent all possible floating point values, some precision will be lost. A float is said to represent *single-precision* floating point whereas a double is said to represent *double-precision* floating point. (Since a double has 64 bits, it can dedicate more bits to both the mantissa and exponent fields, allowing for more precision.)



How does precision play out in practice? The figure above shows how unexpected (or at least unintuitive) things can happen due to imprecision. If you take the square root of 2.0 and store it in the float fRoot you get a particular value. If you store the number in a double, the value is the same number to the 8th decimal place; then the two values diverge. Interestingly, in neither case if you take the square root of 2.0 and square it, do you end up with exactly 2.0. Notice how the code in Figure 3.7 tests to see whether the root of 2.0 squared yields 2.0 and in neither case it does. As we will discuss in the next section, the default print setting for floats and doubles is to print up to 6 decimal places (see figure below). As a consequence, the user has no reason to think that these numbers are not exactly 2.0 and the fact that the neither test for equality to 2.0 passes is simply confusing.

It is important for programmers to understand precision when they choose types for their variables and when they perform tests on variables whose values are assumed to be known. Some programs will need more precision in order to run correctly. Some programs will have to allow for a small degree of imprecision in order to run correctly. Understanding exactly the level of precision required for your code is critical to writing correct code. For every project you begin (or join) it is definitely worth taking a minute to think about the code and how important precision might be in that particular domain. This is true particularly for programs that will ultimately be used to make life-and-death decisions for those who have no say over the precision decisions you are making for them.

It is also important to understand the cost. A double takes up twice as much space as a float. This may not matter for a single variable, but some programs declare thousands or even millions of variables at a time. If these variables do not require the precision of a double, choosing a float can make your code run faster and use less memory with no loss of correctness.

format specifier	will be printed as ...
%c	single character
%d	decimal integer
%u	unsigned decimal number
%o	octal number
%x	unsigned hexadecimal number
%f	decimal floating point
%e	scientific notation
%g	picks the shorter of %e or %f
%s	string (prints chars until '\0')
%%	prints the % character!

decimal formatting	will be printed as ...
%f	default: shows 6 decimal places
%.nf	shows n decimal places
%mf	prints with minimum width m
%m.nf	n decimal places *and* minimum width m

escape sequence	will be printed as ...
\n	newline
\t	tab
\\	prints a backslash