Ch. 02
Arithmetic Precision

Float and Double precision datatype

- *Singles* or *floats* is shorthand for *single- precision floating-point numbers and* occupy 32 bits: 1 bit for the sign, 8 bits for the exponent, and 23 bits for the fractional mantissa:

	s	e		f	
Bit position	31	30	23	22	0

- The sign bit s is in bit position 31, the biased exponent e is in bits 30–23, and the fractional part of the mantissa f is in bits 22–0. Since 8 bits are used to store the exponent e and since $2^8 = 256 \rightarrow 0 \leq e \leq 255$.
- Likewise $-126 \leq e \leq 127$.
- In summary, single-precision (32-bit or 4-byte) numbers have six or seven decimal places of significance and magnitudes in the range

$$1.4 \times 10^{-45} \leq \text{single precision} \leq 3.4 \times 10^{38}$$

Float and Double precision datatype

- Doubles are stored as two 32-bit words, for a total of 64 bits (8 B). The sign occupies 1 bit, the exponent e, 11 bits, and the fractional mantissa, 52 bits:

	s	e		f		f (cont.)	
Bit position	63	62	52	51	32	31	0

- The fields are stored contiguously, with part of the mantissa f stored in separate 32-bit words.
- Doubles have approximately 16 decimal places of precision (1 part in 252) and magnitudes in the range

$$4.9 \times 10^{-324} \leq \text{double precision} \leq 1.8 \times 10^{308}.$$

Overflow and Underflow

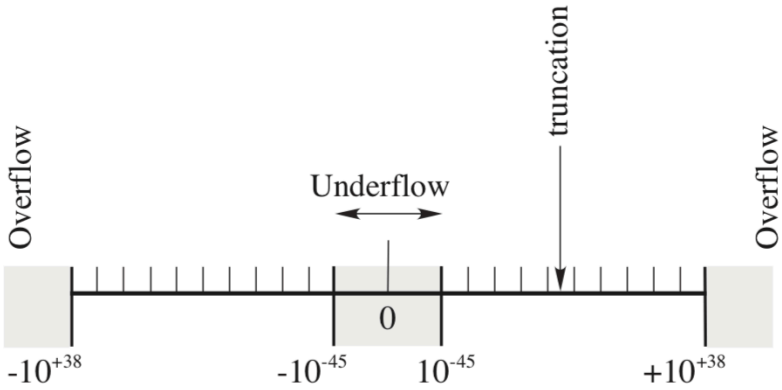

Figure 1.7 The limits of single-precision floating-point numbers and the consequences of exceeding these limits. The hash marks represent the values of numbers that can be stored; storing a number in between these values leads to truncation error. The shaded areas correspond to over- and underflow.

- If a single-precision number $x > 2^{128}$, a fault condition known as an *overflow* occurs. The resulting number x_c may end up being a machine-dependent pattern, not a number (NAN), or unpredictable.
- If $x < 2^{-128}$, an *underflow* occurs. The resulting number x_c is usually set to zero, although this can usually be changed via a compiler option.
- In our experience, *serious scientific calculations almost always require at least 64-bit (double-precision) floats*. And if you need double precision in one part of your calculation, you probably need it all over, which means double-precision library routines for methods and functions.

Determining Machine Precision

- The loss of precision is categorized by defining the *machine precision* ε_m as the maximum positive number that can be added unity without changing it:

$$1_c + \epsilon_m \overset{\text{def}}{=} 1_c,$$

where the subscript *c* is a reminder that this is a computer representation of 1.

- Consequently, an arbitrary number *x* can be thought of as related to its floating- point representation x_c by

$$x_c = x(1 \pm \epsilon), \quad |\epsilon| \leq \epsilon_m,$$

but the actual value for ε is not known.

- In other words, except for powers of 2 that are represented exactly, we should assume that all single-precision numbers contain an error in the sixth decimal place and that all doubles have an error in the fifteenth place.

C and C++ Data-Type Range

- In 1987, the Institute of Electrical and Electronics Engineers (IEEE) and the American National Standards Institute (ANSI) adopted the IEEE 754 standard for floating-point arithmetic. When the standard is followed, you can expect the primitive data types to have the precision and ranges given by the following table

The IEEE 754 Standard for Java's Primitive Data Types

Name	*Type*	*Bits*	*Bytes*	*Range*
boolean	Logical	1	$\frac{1}{8}$	true or false
char	String	16	2	'\u0000' $\leftrightarrow$ '\uFFFF' (ISO Unicode characters)
byte	Integer	8	1	$-128 \leftrightarrow +127$
short	Integer	16	2	$-32,768 \leftrightarrow +32,767$
int	Integer	32	4	$-2,147,483,648 \leftrightarrow +2,147,483,647$
long	Integer	64	8	$-9,223,372,036,854,775,808 \leftrightarrow 9,223,372,036,854,775,807$
float	Floating	32	4	$\pm 1.401298 \times 10^{-45} \leftrightarrow \pm 3.402923 \times 10^{+38}$
double	Floating	64	8	$\pm 4.94065645841246544 \times 10^{-324} \leftrightarrow \pm 1.7976931348623157 \times 10^{+308}$

Practice Session

- Write a computer program to determine the machine precision (`precision.cpp`).
- `quadratic.cpp:` solve quadratic equation (avoid subtractive cancellation between large numbers)
- `roundoff.cpp:` Numerical approximation to sqrt(x^2 + 1) – x and 1 - cos(x)
- `sin_series.cpp`: compute taylor series of sin(x) for x = 1, up to a precision of 10^{-8} (last term in the series should contribute < 10^{-8}) using term by term summation and recurrence relation;
- Iterative schemes:
  - `babyl.cpp`: compute the square root of a number using Babylonian (or Heron's) method (see next page)
  - `unstable_roundoff.cpp`: not all recurrence relation are numerically stable!

Practice Session: Computing the square root

- babyl.cpp:

```
//////////////////////////////////////////////////////////////////////
// Compute the square root using Babylonian method.
// Indeed, finding sqrt(s) is the same as solving the equation
//
//   f(x) = x^2 - s = 0
//
// using Newton method:
//
//  x[n+1] = x[n] - f[n]/dfdx[n] = x[n] - (x[n]^2 - s)/(2*x[n])
//         = (x[n] + S/x[n])/2
//
// The basic idea is that if x is an overestimate to the square root of a
// non-negative real number S then S/x, will be an underestimate and so the
// average of these two numbers may reasonably be expected to provide a
// better approximation.
//
// This is also known as "Heron's method", named after the first-century
// Greek mathematician Heron of Alexandria who gave the first explicit
// description of the method.
// It can be derived from (but predates by 16 centuries) Newton's method.
//
// [source: http://en.wikipedia.org/wiki/Methods_of_computing_square_roots]
//
// For fun only: the fast inverse square root
// (http://en.wikipedia.org/wiki/Fast_inverse_square_root)
// The algorithm was probably developed at Silicon Graphics in the
// early 1990s, and an implementation appeared in 1999 in the Quake III
// Arena source code, but the method did not appear on public forums
// such as Usenet until 2002 or 2003
//////////////////////////////////////////////////////////////////////
```