Essential C++

Numbers

Uwe Naumann

Informatik 12 (STCE), RWTH Aachen University, Germany
naumann@stce.rwth-aachen.de

Topics covered by another (series of) "Essential ..." article(s) are marked by ▶ . Further important terminology is *highlighted*.

Prerequisites

- ▶ Infrastructure
- ▶ First Steps
- ▶ Fundamental Types
- ▶ Expressions and Assignments

Integers

Integers are represented as binary numbers of varying lengths (**char**: 1B(yte), **short**: 2B, **int**: 4B, **long**: 8B), e.g, the declaration

```
int i=42;
```

yields allocation of 4B initialized with $32 + 8 + 2 = 2^5 + 2^3 + 2^1$ as

$$00000000\ 00000000\ 00000000\ 00101010\ .$$

Negative integer values are defined as the two-complement ((~i)+0b1) of the corresponding positive value, e.g, the declaration

```
int i=-42;
```

yields allocation of 4B initialized with as

$$11111111\ 11111111\ 11111111\ 11010110\ .$$

Efficiency of integer arithmetic may benefit from the bitwise operations

~i: bitwise negation (one-complement, e.g, ~i equals -2)

i&j: bitwise and (e.g, 1&2 equals 0)

i|j: bitwise or (e.g, 1|2 equals 3)

i ˆ j : bitwise exclusive or (e.g, 2ˆ3 equals 1)

i <<j: bitwise left shift (e.g, 1<<2 equals 4)

i >>j: bitwise right shift (e.g, 12>>2 equals 3)

Information on ranges covered by integer types are provided by the <limits> chapter of the standard library.

ASCII

Integers of type **char** encode characters according to ASCII[1]. The values between 33 and 126 are printed as

```
! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A B C D E F G H I J K L M N O
P Q R S T U V W X Y Z [ \ ] ^ _ ‘ a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~
```

Further special characters include backspace (8), horizontal tabulator (9) and escape (27). The (potentially larger) type **wchar_t** is used for extensions of ASCII such as Unicode.

Floating-Point Numbers

Real numbers $x \in \mathbb{R}$ are represented as floating-point numbers with base β, precision t and exponent range $[L, U]$ as follows:

$$x = \pm \left(m_0 + \frac{m_1}{\beta} + \frac{m_2}{\beta^2} + \ldots + \frac{m_{t-1}}{\beta^{t-1}} \right) \beta^e$$

where $0 \leq m_i \leq \beta - 1$ for $i = 0, \ldots, t-1$ and $L \leq e \leq U$. The sequences of base-β digits $m = m_0 m_1 \ldots m_{t-1}$ and $e = e_0 e_1 \ldots e_{s-1}$ are called mantissa (also: significant) and exponent. The signed exponent is biased by shifting into the positive range through addition of $2^{s-1} - 1$. Thus comparison of floating-point numbers can be simplified.

All relevant information on floating-point numbers is provided by the <limits> chapter of the standard library including the smallest positive floating-point numbers whose sum with one is greater than one (e.g, numeric_limits<**float**>::epsilon()). This *machine epsilon* ϵ quantifies the limits of the precision of floating-point arithmetic on the given machine with the given data type.

Typically, floating-point numbers are *normalized* as $m_0 = 1$ unless $x = 0$, i.e, $1 \leq m < \beta$. Absolute values below the smallest non-vanishing positive floating-value (e.g, numeric_limits<**float**>::min()) are represented as zero (*underflow*). Hence, division by the difference of two almost equal numbers may lead to division by zero. Absolute values larger than the largest non-vanishing positive floating-value (e.g, std :: numeric_limits<**float**>::max()) result in *overflow*, which can lead to further dramatic numerical errors.

For example, the floating-point number system defined by $\beta = 2$, $t = 3$ and $[L, U] = [-1, 1]$ contains the following 25 elements:

$$\begin{aligned}
&0 \\
&\pm 1.00_2 * 2^{-1} = \pm 0.5_{10}, \quad \pm 1.01_2 * 2^{-1} = \pm 0.625_{10} \\
&\pm 1.10_2 * 2^{-1} = \pm 0.75_{10}, \quad \pm 1.11_2 * 2^{-1} = \pm 0.875_{10} \\
&\pm 1.00_2 * 2^{0} = \pm 1_{10}, \quad \pm 1.01_2 * 2^{0} = \pm 1.25_{10} \\
&\pm 1.10_2 * 2^{0} = \pm 1.5_{10}, \quad \pm 1.11_2 * 2^{0} = \pm 1.75_{10} \\
&\pm 1.00_2 * 2^{1} = \pm 2_{10}, \quad \pm 1.01_2 * 2^{1} = \pm 2.5_{10} \\
&\pm 1.10_2 * 2^{1} = \pm 3_{10}, \quad \pm 1.11_2 * 2^{1} = \pm 3.5_{10}
\end{aligned}$$

[1] American Standard Code for Information Interchange

where subscripts denote the base (binary or decimal) of the given sequence of digits.

Denormalized floating-point numbers mitigate underflow by eliminating the assumption that $m_0 = 1$. The range of the mantissa is modified accordingly.

float

Single-precision floating-point numbers use 23 bits for the mantissa (24th bit equal to 1 due to normlization), 8 bits for the exponent and 1 bit for the sign yielding six significant digits in decimal format with absolute values ranging over $[1.17549e-38, 3.40282e+38]$. Examples relating decimal values to their binary floating-point representation include

$$\begin{aligned} 0 &\hat{=} 00000000\ 00000000\ 00000000\ 00000000 \\ 1 &\hat{=} 00111111\ 10000000\ 00000000\ 00000000 \\ -2.1 &\hat{=} 11000000\ 00000110\ 01100110\ 01100110\ . \end{aligned}$$

The following program prints -2.1 on the screen by interpretation of the corresponding floating-point representation.

Listing 1: Floating-Point Number

```
#include <iostream>
#include <cmath>

int main() {
  std::cout << - // sign
     pow(2,
            pow(2,7) // exponent + 2^7-1 (bias)
            -(pow(2,7)-1) // unbias
          )*(
           1+pow(2,-5)+pow(2,-6)+pow(2,-9)+pow(2,-10)
           +pow(2,-13)+pow(2,-14)+pow(2,-17)+pow(2,-18)
           +pow(2,-21)+pow(2,-22) // mantissa
           )
            << std::endl;
  return 0;
}
```

double

Double-precision floating-point numbers use 52 bits for the mantissa (53rd bit equal to 1 due to normlization), 11 bits for the exponent and 1 bit for the sign yielding fifteen significant digits in decimal format with absolute values ranging over $[2.22507e - 308, 1.79769e + 308]$.

Special Numbers

- 0: all bits equal to zero, e.g., for single precision

  $$00000000\ 00000000\ 00000000\ 00000000$$

- -0: sign bit equal to one; remaining bits equal to zero, e.g,

  $$10000000\ 00000000\ 00000000\ 00000000$$

  (underflow of a negative number)

- ∞: bits of biased exponent equal to one; remaining bits equal to zero, e.g,

$$01111111\ 10000000\ 00000000\ 00000000$$

- $-\infty$: bits of mantissa equal to zero; remaining bits equal to one, e.g,

$$11111111\ 10000000\ 00000000\ 00000000$$

- NaN (not a number): bits of biased exponent equal to one; arbitrary sign; arbitrary non-zero mantissa , e.g,

$$01111111\ 10000000\ 00000100\ 00000000$$

Operations which result in special numbers include

$$\frac{x}{0} = \begin{cases} \infty & x > 0 \\ \text{NaN} & x = 0 \\ -\infty & x < 0 \end{cases}$$

$$0 \cdot \infty = \text{NaN} \quad x < 0 \; .$$

Numerical Issues

Floating-point values form a grid. Most real values cannot be represented exactly. They are typically *rounded* to the nearest representable value, e.g, $1.126 \approx 1.25$ in $(\beta = 2, t = 3, [L, U] = [-1, 1])$. Subtraction of two almost equal numbers with differences limited to the last k digits of the mantissa yields a result with and accuracy of only k digits. This effect is known as *cancellation*.

Combinations of rounding and cancellation can lead to potentially dramatic errors in numerical computations. Finte difference approximation of first (and higher) derivatives of differentiable functions $y = f(x)$ implemented as computer programs represents a famous example. Building on the definition of the derivative of f as

$$\frac{df}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x-h)}{2 \cdot h}$$

central finite differences approximate the derivative as

$$\frac{df}{dx} \approx \frac{f(x+h) - f(x-h)}{2 \cdot h}$$

for a suitable h. Choosing a "suitable" h can be tricky as its absolute value cannot be decreased arbitrarly in floating-point arithmetic. The following table lists the results obtained for $y = x^3$ at $x = 1$ for $h = 1, \ldots, 10^{-16}$. Obviously, the correct result is equal to 3.

h	**float**	**double**
10^{0}	4	4
10^{-1}	3.01	3.01
10^{-2}	3.0001	3.0001
10^{-3}	3.00005	3
$\mathbf{3.45267 \cdot 10^{-4}}$	**2.99994**	3
10^{-4}	3.0002	3
10^{-5}	3.00407	3
10^{-6}	2.95043	3
10^{-7}	3.57628	3
$\mathbf{1.49012 \cdot 10^{-8}}$	0	**3**
10^{-8}	0	3
10^{-9}	0	3
10^{-10}	0	3
10^{-11}	0	3.0001
10^{-12}	0	2.99927
10^{-13}	0	2.9976
10^{-14}	0	3.16414
10^{-15}	0	1.66533
10^{-16}	0	0

A suitable h needs to be compromise between accuracy (small h) and numerical stability (not too small h) of the approximation. Various mathematical properties of f impact the choice. A rule of thumb suggests a perturbation of $x = 1$ at the center of its mantissa, which is obtained by setting $h = \sqrt{\epsilon}$. The corresponding entries for **float** and **double** are printed in bold The following sample program ilustrates this approach.

Listing 2: Numerical Differentiation

```
#include <cmath>
#include <limits>
#include <iostream>

using T=float; // replace T with float from here onwards

T f(T x) { return pow(x,3); }

int main() {
  T x=1, h=sqrt(std::numeric_limits<T>::epsilon());
  std::cout << (f(x+h)-f(x-h))/(2*h) << std::endl;
  return 0;
}
```

It produces the output 2.99994.

References

[1] https://www.cppreference.com.

[2] https://docs.microsoft.com/en-us/cpp/cpp.

[3] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 4th edition, 2013.