Computer Arithmetic

Uwe Naumann

Informatik 12:
Software and Tools for Computational Engineering

RWTH Aachen University

Contents

Objective and Learning Outcomes

Computer Arithmetic
- Motivation
- Integer Numbers
- Floating-Point Numbers

Summary and Next Steps

Outline

Objective and Learning Outcomes

Computer Arithmetic
- Motivation
- Integer Numbers
- Floating-Point Numbers

Summary and Next Steps

Objective

- Introduction to fundamental numeric data types and computer arithmetic

Learning Outcomes

- You will understand
  - integer and floating-point number representation
  - limitations of computer arithmetic.
- You will be able to
  - convert decimal into binary numbers and vice versa
  - use auxiliary C++ function to_bin for automatic conversion into binary format.

Outline

Objective and Learning Outcomes

Computer Arithmetic
- Motivation
- Integer Numbers
- Floating-Point Numbers

Summary and Next Steps

Computer Arithmetic

Motivation


```cpp
#include<iostream>

int main() {
  for (int n=1;n<=10;n++) {
    float dt=1./n, t=0;
    for (int i=0;i<n;i++,t+=dt);
    for (int i=0;i<n;i++,t-=dt);
    std::cout << t << std::endl;
  }
  return 0;
}
```

```
0
0
-5.96046e-08
0
2.98023e-08
-8.9407e-08
2.98023e-08
0
4.47035e-08
1.49012e-08
```

Integer numbers are coded as binary numbers.

For example, 8 becomes 00000000 00000000 00000000 00001000.

Negative integer numbers are coded as the two complements of the corresponding positive binary numbers. Two complements are obtained by adding one to the one complement. The latter is build by switching individual digits $0 \leftrightarrow 1$.

For example, -8 becomes 11111111 11111111 11111111 11111000.

Our C++ function `to_bin(T)` prints the binary representation of a value of data type `T`. It can be used to study the binary representation of integers.

Integer Numbers

C++ Standard Library Support: <limits>

Software and Tools for Computational Engineering

RWTH AACHEN UNIVERSITY

The C++ Standard Library provides support for querying the characteristics of integers as well as of other numeric data types.

```
#include<iostream>
#include<limits>
using namespace std;

int main() {
  cout << numeric_limits<int>::is_exact << endl; // 1
  cout << numeric_limits<int>::max() << endl; // 2147483647
  cout << numeric_limits<int>::min() << endl; // -2147483648
  cout << numeric_limits<int>::digits << endl; // 31
  cout << numeric_limits<int>::digits10 << endl; // 9
  return 0;
}
```

Binary representations of appropriate values (e.g, max) can be generated by to_bin, e.g.

```
  to_bin(numeric_limits<int>::max());
```

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

$$x = \pm \left(d_0 + \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \ldots + \frac{d_{t-1}}{\beta^{t-1}} \right) \beta^e \,,$$

where $0 \leq d_i \leq \beta - 1$ for $i = 0, \ldots, t-1$ and $L \leq e \leq U$.

The sequence of digits $m = d_0 d_1 \ldots d_{t-1}$ over base β is called mantissa. The exponent is denoted as e.

We consider normalized floating-point number systems, that is, $d_0 = 0 \Leftrightarrow x = 0$ implying $1 \leq m < \beta$.

Decimal Floating-Point Numbers

Example

The normalized floating-point number system defined by $\beta = 10$, $t = 3$ and $[L, U] = [-2, 2]$ contains for example

$$\pm 0.0127 = \pm\left(1 + \frac{2}{10} + \frac{7}{10^2}\right) \cdot 10^{-2} = \pm 1.27 \cdot 10^{-2}$$

$$\pm 98.1 = \pm\left(9 + \frac{8}{10} + \frac{1}{10^2}\right) \cdot 10^{1} = \pm 9.81 \cdot 10$$

$$\pm 1 = \pm\left(1 + \frac{0}{10} + \frac{0}{10^2}\right) \cdot 10^{0} = \pm 1.00 \cdot 1$$

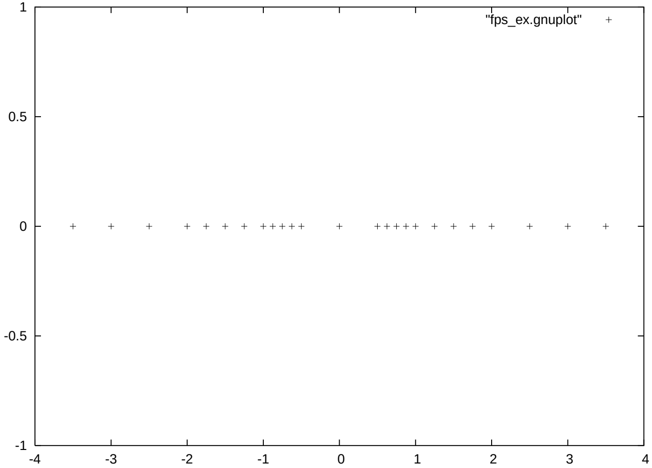
Minimum absolute value: $0.01 = 1.00 \cdot 10^{-2} \Rightarrow$ underflow

Maximum absolute value: $999 = 9.99 \cdot 10^{2} \Rightarrow$ overflow

The normalized floating-point number system defined by $\beta = 2$, $t = 3$ and $[L, U] = [-1, 1]$ consists of the following 25 elements:

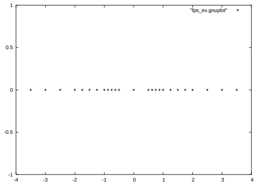
$$
\begin{aligned}
&0\\
&\pm 1.00_2 * 2^{-1} = \pm 0.5_{10}, \quad \pm 1.01_2 * 2^{-1} = \pm 0.625_{10}\\
&\pm 1.10_2 * 2^{-1} = \pm 0.75_{10}, \quad \pm 1.11_2 * 2^{-1} = \pm 0.875_{10}\\
&\pm 1.00_2 * 2^{0} = \pm 1_{10}, \quad \pm 1.01_2 * 2^{0} = \pm 1.25_{10}\\
&\pm 1.10_2 * 2^{0} = \pm 1.5_{10},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

Example: $\beta = 2, t = 3, [L, U] = [-1, 1]$



Real values which cannot be represented exactly within the given floating-point number system are typically rounded to the nearest representable number.

Ties are broken by rounding to even mantissa (last bit of mantissa equal to zero).



Example: In $(\beta = 2, t = 3, [L, U] = [-1, 1])$ we get

$$1.126 \approx 1.25$$

and

$$1.125 \approx 1.$$

... uses 32 bits:

- 23 bits for the mantissa (24th bit equal to 1 due to normalization)
- 8 bits for the exponent
- 1 sign bit

yielding 6 significant digits in decimal output format with minimum and maximum absolute values of 1.17549e-38 and 3.40282e+38, respectively.

The signed exponent is shifted into the $1 \dots 254$ range (biased exponent). Its true value is obtained by subtracting $2^7 - 1 = 127$. This bias replaces the traditional two's complement format for negative integers in order to simplify the comparison of two values of data type **float**.

See IEEE 754 standard for further details.

Our C++ function to_bin(T) prints binary representation of value of data type T.

```
to_bin(static_cast<float>(1.0));
cout << pow(2,
                pow(2,0)+pow(2,1)+pow(2,2)+pow(2,3)+pow(2,4)
                +pow(2,5)+pow(2,6) // exponent + 2^7-1 (bias)
                -(pow(2,7)-1) // correction of bias
         )
         *1 // mantissa
         << endl;
```

yields output

```
00111111 10000000 00000000 00000000
1
```

Note: floating-point constants are interpeted as of type **double** by default, e.g, to_bin(1.0) generates the output

```
00111111 11110000 00000000 00000000 00000000 00000000 00000000 00000000
```

Floating-Point Numbers

static_cast<float>(2.1)

```
to_bin(static_cast<float>(2.1));
cout << pow(2,
            pow(2,7) // exponent + 2^7-1 (bias)
            -(pow(2,7)-1) // correction of bias
           )
           *(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
     << endl;
```

yields output

```
01000000 00000110 01100110 01100110
2.1
```

```
to_bin(static_cast<float>(0.0));
cout << pow(2,0 // exponent + 2^7-1 (bias)
              -(pow(2,7)-1)// correction of bias
             )
              *1 // mantissa
      << endl;
```

yields output

```
00000000 00000000 00000000 00000000
5.87747e-39
```

All floating-point numbers are distinct from zero due to normalization. The floating-point number with all bits vanished is explicitly defined as zero.

Further special numbers are infinity $\pm$inf ($e = 2^8 - 1,\ m = 0$) and "not a number" nan ($e = 2^8 - 1,\ m > 0$).

Floating-Point Numbers

<limits>

The C++ Standard Library provides support for querying the characteristics of **float** as well as of other numeric data types.

```
#include<iostream>
#include<limits>
using namespace std;

int main() {
  cout << numeric_limits<float>::is_exact << endl; // 0
  cout << numeric_limits<float>::epsilon() << endl; // 1.19209e−07
  cout << numeric_limits<float>::max() << endl; // 3.40282e+38
  cout << numeric_limits<float>::min() << endl; // 1.17549e−38
  cout << numeric_limits<float>::lowest() << endl; // −3.40282e+38
  cout << numeric_limits<float>::digits << endl; // 24
  cout << numeric_limits<float>::min_exponent << endl; // −125
  cout << numeric_limits<float>::max_exponent << endl; // 128
  return 0;
}
```

... uses 64 bits:

- 52 bits for the mantissa (53rd bit equal to 1 due to normalization)
- 11 bits for the exponent
- 1 sign bit

yielding 15 significant digits in decimal output format with minimum and maximum absolute values of 2.22507e-308 and 1.79769e+308, respectively.

The signed exponent is shifted into the $1 \ldots 2046$ range (biased exponent). Its true value is obtained by subtracting $2^{10} - 1 = 1023$.

See IEEE 754 standard for further details.

If two floating-point numbers x and y are identical except for the last k digits of their mantissas, then $z = x - y$ exhibits only k digits precision, which can have a negative impact on the accuracy of subsequent computations.

For example,

```
double h=1e-13,a=1,b,c,d;
b=a+h; c=b-a; d=c/h;
std::cout << h << " " << a << " " << b << " " << c << " " << d << std::endl;
```

yields the output

```
1e-13 1 1 9.99201e-14 0.999201
```

`d` should be equal to 1.

Conclusion: Use floating-point arithmetic with care!

Outline

Objective and Learning Outcomes

Computer Arithmetic
- Motivation
- Integer Numbers
- Floating-Point Numbers

Summary and Next Steps

Summary

- binary representation of integers and reals
- single and double precision floating-point arithmetic
- issues: rounding, cancelation, over-/underflow

Next Steps

- Investigate numeric data types using to_bin.
- Consult IEEE 754 standard.
- Continue the course to find out more ...