2.5 — Floating point numbers

BY ALEX ON JUNE 9TH, 2007 | LAST MODIFIED BY ALEX ON NOVEMBER 8TH, 2017

Integers are great for counting whole numbers, but sometimes we need to store *very* large numbers, or numbers with a fractional component. A **floating point** type variable is a variable that can hold a real number, such as 4320.0, -3.33, or 0.01226. The *floating* part of the name *floating point* refers to the fact that the decimal point can “float”; that is, it can support a variable number of digits before and after the decimal point.

There are three different floating point data types: **float**, **double**, and **long double**. As with integers, C++ does not define the size of these types. On modern architectures, floating point representation almost always follows IEEE 754 binary format. In this format, a float is 4 bytes, a double is 8, and a long double can be equivalent to a double (8 bytes), 80-bits (often padded to 12 bytes), or 16 bytes.

Floating point data types are always signed (can hold positive and negative values).

Category	Type	Minimum Size	Typical Size
floating point	float	4 bytes	4 bytes
	double	8 bytes	8 bytes
	long double	8 bytes	8, 12, or 16 bytes

Here are some definitions of floating point numbers:

```cpp
float fValue;
double dValue;
long double dValue2;
```

When using floating point literals, it is convention to always include at least one decimal place. This helps distinguish floating point values from integer values.

```cpp
int x(5); // 5 means integer
double y(5.0); // 5.0 is a floating point literal (no suffix means double type by default)
float z(5.0f); // 5.0 is a floating point literal, f suffix means float type
```

Note that by default, floating point literals default to type double. An f suffix is used to denote a literal of type float.

Scientific notation

How floating point variables store information is beyond the scope of this tutorial, but it is very similar to how numbers are written in scientific notation. **Scientific notation** is a useful shorthand for writing lengthy numbers in a concise manner. And although scientific notation may seem foreign at first, understanding scientific notation will help you understand how floating point numbers work, and more importantly, what their limitations are.

Numbers in scientific notation take the following form: *significand* x $10^{exponent}$. For example, in the scientific notation `1.2 x` 10^4, `1.2` is the significand and 4 is the exponent. This number evaluates to 12,000.

By convention, numbers in scientific notation are written with one digit before the decimal, and the rest of the digits afterward.

Consider the mass of the Earth. In decimal notation, we’d write this as `5973600000000000000000000 kg`. That’s a really large number (too big to fit even in an 8 byte integer). It’s also hard to read (is that 19 or 20 zeros?). In scientific notation, this would be written as `5.9736 x` 10^{24} `kg`, which is much easier to read. Scientific notation has the added benefit of making it easier to compare the magnitude of two really large or really small numbers simply by comparing the exponent.

Because it can be hard to type or display exponents in C++, we use the letter ‘e’ or ‘E’ to represent the “times 10 to the power of” part of the equation. For example, `1.2 x` 10^4 would be written as `1.2e4`, and `5.9736 x` 10^{24} would be written as `5.9736e24`.

For numbers smaller than 1, the exponent can be negative. The number `5e-2` is equivalent to `5 *` 10^{-2}, which is `5 /` 10^2, or `0.05`. The mass of an electron is `9.1093822e-31 kg`.

In fact, we can use scientific notation to assign values to floating point variables.

```cpp
double d1(5000.0);
double d2(5e3); // another way to assign 5000

double d3(0.05);
double d4(5e-2); // another way to assign 0.05
```

How to convert numbers to scientific notation

Use the following procedure:

- Your exponent starts at zero.
- Slide the decimal so there is only one non-zero digit to the left of the decimal.
  - Each place you slide the decimal to the left increases the exponent by 1.
  - Each place you slide the decimal to the right decreases the exponent by 1.
- Trim off any leading zeros (on the left end)
- Trim off any trailing zeros (on the right end) only if the original number had no decimal point. We're assuming they're not significant unless otherwise specified.

Here's some examples:

```
Start with: 42030
Slide decimal left 4 spaces: 4.2030e4
No leading zeros to trim: 4.2030e4
Trim trailing zeros: 4.203e4 (4 significant digits)


Start with: 0.0078900
Slide decimal right 3 spaces: 0007.8900e-3
Trim leading zeros: 7.8900e-3
Don't trim trailing zeros: 7.8900e-3 (5 significant digits)


Start with: 600.410
Slide decimal left 2 spaces: 6.00410e2
No leading zeros to trim: 6.00410e2
Don't trim trailing zeros: 6.00410e2 (6 significant digits)
```

Here's the most important thing to understand: The digits in the significand (the part before the E) are called the **significant digits**. The number of significant digits defines a number's **precision**. The more digits in the significand, the more precise a number is.

Precision and trailing zeros after the decimal

Consider the case where we ask two lab assistants each to weigh the same apple. One returns and says the apple weighs 87 grams. The other returns and says the apple weighs 87.000 grams. Assuming the weighings were correct, in the former case, we know the apple actually weighs somewhere between 86.50 and 87.49 grams. Maybe the scale was only precise to the nearest gram. Or maybe our assistant rounded a bit. In the latter case, we are confident about the actual weight of the apple to a much higher degree (it weighs between 86.9950 and 87.0049 grams, which has much less variability).

So in scientific notation, we prefer to keep trailing zeros after a decimal, because those digits impart useful information about the precision of the number.

However, in C++, 87 and 87.000 are treated exactly the same, and the compiler will store the same value for each. There's no technical reason why we should prefer one over the other (though there might be scientific reasons, if you're using the source code as documentation).

Precision and range

Consider the fraction 1/3. The decimal representation of this number is 0.33333333333333… with 3's going out to infinity. An infinite length number would require infinite memory to store, and we typically only have 4 or 8 bytes. Floating point numbers can only store a certain number of significant digits, and the rest are lost. The **precision** of a floating point number defines how many *significant digits* it can represent without information loss.

When outputting floating point numbers, std::cout has a default precision of 6 -- that is, it assumes all floating point variables are only significant to 6 digits, and hence it will truncate anything after that.

The following program shows std::cout truncating to 6 digits:

```cpp
#include <iostream>
int main()
{
    float f;
    f = 9.87654321f; // f suffix means this number should be treated as a float
    std::cout << f << std::endl;
    f = 987.654321f;
    std::cout << f << std::endl;
    f = 987654.321f;
    std::cout << f << std::endl;
    f = 9876543.21f;
    std::cout << f << std::endl;
    f = 0.0000987654321f;
    std::cout << f << std::endl;
    return 0;
}
```

This program outputs:

```
9.87654
987.654
987654
9.87654e+006
9.87654e-005
```

Note that each of these is only 6 significant digits.

Also note that cout will switch to outputting numbers in scientific notation in some cases. Depending on the compiler, the exponent will typically be padded to a minimum number of digits. Fear not, 9.87654e+006 is the same as 9.87654e6, just with some padding 0's. The minimum number of exponent digits displayed is compiler-specific (Visual Studio uses 3, some others use 2 as per the C99 standard).

However, we can override the default precision that cout shows by using the std::setprecision() function that is defined in a header file called iomanip.

```cpp
#include <iostream>
#include <iomanip> // for std::setprecision()
int main()
{
    std::cout << std::setprecision(16); // show 16 digits
    float f = 3.33333333333333333333333333333333333333f;
    std::cout << f << std::endl;
    double d = 3.3333333333333333333333333333333333333;
    std::cout << d << std::endl;
    return 0;
}
```

Outputs:

```
3.333333253860474
3.333333333333334
```

Because we set the precision to 16 digits, each of the above numbers is printed with 16 digits. But, as you can see, the numbers certainly aren't precise to 16 digits!

The number of digits of precision a floating point variable has depends on both the size (floats have less precision than doubles) and the particular value being stored (some values have more precision than others). Float values have between 6 and 9 digits of precision, with most float values having at least 7 significant digits (which is why everything after that many digits in our answer above is junk). Double values have between 15 and 18 digits of precision, with most double values having at least 16 significant digits. Long double has a minimum precision of 15, 18, or 33 significant digits depending on how many bytes it occupies.

Precision issues don't just impact fractional numbers, they impact any number with too many significant digits. Let's consider a big number:

```
#include <iostream>
#include <iomanip> // for std::setprecision()

int main()
{
    float f(123456789.0f); // f has 10 significant digits
    std::cout << std::setprecision(9); // to show 9 digits in f
    std::cout << f << std::endl;
    return 0;
}
```

Output:

```
123456792
```

123456792 is greater than 123456789. The value 123456789.0 has 10 significant digits, but float values typically have 7 digits of precision. We lost some precision!

Consequently, one has to be careful when using floating point numbers that require more precision than the variables can hold.

Assuming IEEE 754 representation:

Size	Range	Precision
4 bytes	±1.18 x 10^{-38} to ±3.4 x 10^{38}	6-9 significant digits, typically 7
8 bytes	±2.23 x 10^{-308} to ±1.80 x 10^{308}	15-18 significant digits, typically 16
80-bits (12 bytes)	±3.36 x 10^{-4932} to ±1.18 x 10^{4932}	18-21 significant digits
16 bytes	±3.36 x 10^{-4932} to ±1.18 x 10^{4932}	33-36 significant digits

It may seem a little odd that the 12-byte floating point number has the same range as the 16-byte floating point number. This is because they have the same number of bits dedicated to the exponent -- however, the 16-byte number offers a much higher precision.

Rule: Favor double over float unless space is at a premium, as the lack of precision in a float will often lead to inaccuracies.

Rounding errors

One of the reasons floating point numbers can be tricky is due to non-obvious differences between binary (how data is stored) and decimal (how we think) numbers. Consider the fraction 1/10. In decimal, this is easily represented as 0.1, and we are used to thinking of 0.1 as an easily representable number. However, in binary, 0.1 is represented by the infinite sequence: 0.00011001100110011… Because of this, when we assign 0.1 to a floating point number, we'll run into precision problems.

You can see the effects of this in the following program:

```
#include <iostream>
#include <iomanip> // for std::setprecision()

int main()
{
    double d(0.1);
    std::cout << d << std::endl; // use default cout precision of 6
    std::cout << std::setprecision(17);
    std::cout << d << std::endl;
    return 0;
}
```

This outputs:

```
0.1
0.10000000000000001
```

On the top line, cout prints 0.1, as we expect.

On the bottom line, where we have cout show us 17 digits of precision, we see that d is actually *not quite* 0.1! This is because the double had to truncate the approximation due to its limited memory, which resulted in a number that is not exactly 0.1. This is called a **rounding error**.

Rounding errors can have unexpected consequences:

```cpp
#include <iostream>
#include <iomanip> // for std::setprecision()

int main()
{
    std::cout << std::setprecision(17);

    double d1(1.0);
    std::cout << d1 << std::endl;

    double d2(0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1); // should equal 1.0
    std::cout << d2 << std::endl;
}
```

```
1
0.99999999999999989
```

Although we might expect that d1 and d2 should be equal, we see that they are not. If we were to compare d1 and d2 in a program, the program would probably not perform as expected. We discuss this more in section **3.5 -- Relational operators (comparisons)**.

One last note on rounding errors: mathematical operations (such as addition and multiplication) tend to make rounding errors grow. So even though 0.1 has a rounding error in the 17th significant digit, when we add 0.1 ten times, the rounding error has crept into the 16th significant digit.

NaN and Inf

There are two special categories of floating point numbers. The first is **Inf**, which represents infinity. Inf can be positive or negative. The second is **NaN**, which stands for “Not a Number”. There are several different kinds of NaN (which we won’t discuss here).

Here’s a program showing all three:

```cpp
#include <iostream>

int main()
{
    double zero = 0.0;
    double posinf = 5.0 / zero; // positive infinity
    std::cout << posinf << std::endl;

    double neginf = -5.0 / zero; // negative infinity
    std::cout << neginf << std::endl;

    double nan = zero / zero; // not a number (mathematically invalid)
    std::cout << nan << std::endl;

    return 0;
}
```

And the results using Visual Studio 2008 on Windows:

```
1.#INF
-1.#INF
1.#IND
```

INF stands for infinity, and IND stands for indeterminate. Note that the results of printing Inf and NaN are platform specific, so your results may vary.

Conclusion

To summarize, the two things you should remember about floating point numbers:

1) Floating point numbers are great for storing very large or very small numbers, including those with fractional components, so long as they have a limited number of significant digits (precision).

2) Floating point numbers often have small rounding errors, even when the number has fewer significant digits than the precision. Many times these go unnoticed because they are so small, and because the numbers are truncated for output. Consequently, comparisons of floating point numbers may not give the expected results. Performing mathematical operations on these values will cause the rounding errors to grow larger.

Quiz

1) Convert the following numbers to C++ style scientific notation (using an e to represent the exponent) and determine how many significant digits each has (keep trailing zeros after the decimal):
a) 34.50
b) 0.004000
c) 123.005
d) 146000
e) 146000.001
f) 0.0000000008
g) 34500.0

Quiz Answers

1) **Hide Solution**

a) 3.450e1 (4 significant digits)
b) 4.000e-3 (4 significant digits)
c) 1.23005e2 (6 significant digits)
d) 1.46e5 (3 significant digits)
e) 1.46000001e5 (9 significant digits)

And now a couple of trickier ones:

f) 8e-10 (1 significant digit)

The correct significand is 8, not 8.0. 8.0 has two significant digits, but this number only has 1.

g) 3.45000e4 (6 significant digits)

We don't trim the trailing zeros here because the number *does* have a decimal point. Even though the decimal point doesn't affect the value of the number, it affects the precision, so it needs to be included in the significand. If the number had been specified as 34500, then the answer would have been 3.45e4.

2.6 -- Boolean values and an introduction to if statements

Index

2.4a -- Fixed-width integers and the unsigned controversy