4.8 — Floating point numbers

BY ALEX ON JUNE 9TH, 2007 | LAST MODIFIED BY ALEX ON AUGUST 4TH, 2019

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 actual size of these types (but it does guarantee minimum sizes). 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:

```
1 float fValue;
2 double dValue;
3 long double ldValue;
```

When using floating point literals, always include at least one decimal place (even if the decimal is 0). This helps the compiler understand that the number is a floating point number and not an integer.

```
1 int x{5}; // 5 means integer
2 double y{5.0}; // 5.0 is a floating point literal (no suffix means double type by default)
3 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.

Best practice

Always make sure the type of your literals match the type of the variables they're being assigned to or used to initialize. Otherwise an unnecessary conversion will result, possibly with a loss of precision.

Warning

Make sure you don't use integer literals where floating point literals should be used. This includes when initializing or assigning values to floating point objects, doing floating point arithmetic, and calling functions that expect floating point values.

Printing floating point numbers

Now consider this simple program:

```
#include <iostream>

int main()
{
    std::cout << 5.0 << '\n';
    std::cout << 6.7f << '\n';
    std::cout << 9876543.21 << '\n';
}
```

The results of this seemingly simple program may surprise you:

```
5
6.7
9.87654e+06
```

In the first case, the std::cout printed 5, even though we typed in 5.0. By default, std::cout will not print the fractional part of a number if the fractional part is 0.

In the second case, the number prints as we expect.

In the third case, it printed the number in scientific notation (if you need a refresher on scientific notation, see lesson **4.7 -- Introduction to scientific notation**).

Floating point range

Assuming IEEE 754 representation:

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

The 80-bit floating point type is a bit of a historical anomaly. On modern processors, it is typically implemented using 12 or 16 bytes (which is a more natural size for processors to handle).

It may seem a little odd that the 80-bit floating point type has the same range as the 16-byte floating point type. This is because they have the same number of bits dedicated to the exponent -- however, the 16-byte number can store more significant digits.

Floating point precision

Consider the fraction 1/3. The decimal representation of this number is 0.33333333333333... with 3's going out to infinity. If you were writing this number on a piece of paper, your arm would get tired at some point, and you'd eventually stop writing. And the number you were left with would be close to 0.3333333333.... (with 3's going out to infinity) but not exactly.

On a computer, an infinite length number would require infinite memory to store, and typically we only have 4 or 8 bytes. This limited memory means floating point numbers can only store a certain number of significant digits -- and that any additional significant digits are lost. The number that is actually stored will be close to the desired number, but not exact.

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 (the minimum precision of a float), and hence it will truncate anything after that.

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

```
#include <iostream>
int main()
{
    std::cout << 9.87654321f << '\n';
    std::cout << 987.654321f << '\n';
    std::cout << 987654.321f << '\n';
    std::cout << 9876543.21f << '\n';
    std::cout << 0.0000987654321f << '\n';

    return 0;
}
```

This program outputs:

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

Note that each of these only have 6 significant digits.

Also note that std::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).

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. 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.

We can override the default precision that std::cout shows by using the std::setprecision() function that is defined in the *iomanip* header.

```
#include <iostream>
#include <iomanip> // for std::setprecision()
int main()
{
    std::cout << std::setprecision(16); // show 16 digits of precision
    std::cout << 3.33333333333333333333333333333333333333f <<'\n'; // f suffix means float
    std::cout << 3.33333333333333333333333333333333333333 << '\n'; // no suffix means double
    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! And because floats are less precise than doubles, the float exhibits has more error.

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 << '\n';
    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 (and the result of 123456792 is precise only to 7 significant digits). We lost some precision! When precision is lost because a number can't be stored precisely, this is called a **rounding error**.

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

Best practice

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 make floating point comparisons tricky

Floating point numbers are tricky to work with 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 with 1 significant digit. However, in binary, 0.1 is represented by the infinite sequence: 0.0001100110011... 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 << '\n'; // use default cout precision of 6
    std::cout << std::setprecision(17);
    std::cout << d << '\n';
    return 0;
}
```

This outputs:

```
0.1
0.10000000000000001
```

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

On the bottom line, where we have std::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. The result is a number that is precise to 16 significant digits (which type double guarantees), but the number is not *exactly* 0.1. Rounding errors may make a number either slightly smaller or slightly larger, depending on where the truncation happens.

Rounding errors can have unexpected consequences:

```
#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. Because floating point numbers tend to be inexact, comparing floating point numbers is generally problematic -- we discuss the subject more (and solutions) in lesson **5.6 -- Relational operators and floating point 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. Continued operations would cause this error to become increasingly significant.

Key insight

Rounding errors occur when a number can't be stored precisely. This can happen even with simple numbers, like 0.1. Therefore, rounding errors can, and do, happen all the time. Rounding errors aren't the exception -- they're the rule. Never assume your floating point numbers are exact.

A corollary of this rule is: never use floating point numbers for financial or currency data.

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:

```
#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 useful for storing very large or very small numbers, including those with fractional components.

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. However, 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.

 **4.9 -- Boolean values**

 **Index**

 **4.7 -- Introduction to scientific notation**

C++ TUTORIAL | PRINT THIS POST

362 comments to 4.8 — Floating point numbers

« Older Comments 1 ... 3 4 5

akil
February 8, 2020 at 12:06 pm · Reply

Hi,
There are three different floating point data types: float, double, and long double. As with integers, C++ does not define the actual size of these types (but it does guarantee minimum sizes). What do you mean by (but it does guarantee minimum sizes) ??

scarlet johnson
February 8, 2020 at 9:24 am · Reply

```
#include <iostream>

int main()
{
    float f=0.67;
    if(f == 0.67)
    printf("yes");
    else
    printf("no");
    return 0;
}
```

This program on compiling producing no. Explain Please!

Scarlet Johnson
February 8, 2020 at 10:38 am · Reply

Please ignore my question.

Steve Roy
February 8, 2020 at 6:19 am · Reply

Hii, can you explain me the significance of the suffix f after floating point numbers. And why we are using them while initializing float type variables and cout statements.

nascardriver
February 8, 2020 at 7:33 am · Reply

Literals have types too. '0' is a char, 0 is an integer, 0.0 is a double, 0.0f is a float.

Andrei
February 2, 2020 at 2:59 am · Reply

Why the hell we're setting precision like this??

```
1 std::cout << std::setprecision(9);
```

Until now we where using std::cout with operator<< to OUTPUT something. And code above looks like we're trying to output the result of the function! Why no output happens? (Some people in comments even go with endl after: std::cout << std::setprecision(8) << std::endl;)
Thank you in advance!

nascardriver
February 2, 2020 at 3:28 am · Reply

`std::cout` can do what it wants with the value you give it. For the fundamental types, and types for which this functionality is added later (eg. std::string), it prints the value.
If you find this confusing, you can also use

```
1 std::cout.precision(9);
```

giang
January 6, 2020 at 6:43 pm · Reply

In the d1{1.0} and d2{0.1+....} example above, I still can't understand why the screen shows 1 (even d1 is setprecision to 17 digits), and why d2 gets a result of 0.99999999...?? Can anybody help me explain this situation??

Alex
January 8, 2020 at 12:03 pm · Reply

Because d1 doesn't have precision errors, but d2 does. 0.1 can't be stored precisely as a double (it's like 1/3rd is as a decimal value), so every time we use 0.1 we accrue a little bit of precision loss.

Chrystian Żuberek
December 22, 2019 at 1:14 pm · Reply

From where are taken this random digits that are added to our number after breaching precision?

nascardriver
December 23, 2019 at 6:56 am · Reply

They're caused by the way the numbers are stored. Read up on how floating point numbers are stored in binary if you're interested in where exactly those numbers come from.

Xavier
January 16, 2020 at 1:05 pm · Reply

But I thought floats are stored by separating the exponent and the significant digits? Like the number 1.97, the exponent is -2, and the significant digits will be 197. So if we have the number 0.1, the exponent will be -1, and the significant digits will be 1. I thought floats are not stored by directly storing the fractional binary version of the decimal. Did I miss something?

Cypeace
December 19, 2019 at 9:42 am · Reply

Hi Alex and Nascar,

I've been fiddling around some elementary school math problems and I get somehow unexpected behavior from this function:

```
1  void decimalToFraction(const double decimal) {
2      // first I had problem with "0.52" which then resolved itself but now problem is with "
3      // some numbers work as intended, but some are just creating utter garbage and I cannot
4
5      int     base        { static_cast<int>(decimal) };
6      double  numerator   { decimal };
7      int     denominator { 1 };
8
9      while (numerator - base < 1 && numerator - base > 0) {
10         numerator *= 10;
11         denominator *= 10;
12         base = static_cast<int>(numerator);
13         if (numerator - base == 0) {
14             break;
15         };
16     }
17     std::cout << decimal << " = " << numerator << '/' << denominator << '\n';
18 }
```

Can you please help me identify where am I going wrong?

Thank you very much!
PS: Merry christmass!!

nascardriver
December 20, 2019 at 1:12 am · Reply

Line 13 is unlikely to happen. Floating point values have a limited accuracy, this is covered in detail in chapter 5. Don't compare floating points with equality. You can use `std::abs` (Removes the negative sign, if any) to check if the result is very close to 0.

```
1  if (std::abs(numerator - base) <= 0.0001)
```

There's also a lesson about converting floating points to fractions later.

HolzstockG
December 17, 2019 at 9:59 pm · Reply

Can someone explain me why in computers everything (I know that's an exaggeration) is coming from the power of the number 2. Is it associated with Boolean's algebra?

Also you have mentioned that we should omit using floating point data types for currencies calculators etc. So what should we use instead?

Thanks in advance for reply.

nascardriver
December 18, 2019 at 5:16 am · Reply

Power can be on (High) or off (Low). Each bit can be on (1) or off (0). That's easy to build and easy to work with.

Express the number as an integer. There's a lesson about a fixed precision floating point number later on.

Sri charan Battu
December 11, 2019 at 8:00 am · Reply

Hi, How did 5.0/0.0 return inf? Is it because the values used are float?. When I tried to print 5/0, the program terminated and main returned some bad value.

nascardriver
December 11, 2019 at 8:06 am · Reply

Both variants produce undefined behavior. 5.0 and 0.0 are doubles. If your machine supports inf in double, that can be a result too.

hellmet
November 8, 2019 at 4:55 am · Reply

So std::setprecision only changes how the result is _displayed_, not the actual internal representation, correct? (By internal representation, I mean the IEEE bits representation. 1+11+52 bits for storing the number)

nascardriver
November 9, 2019 at 4:03 am · Reply

Correct. The internal format is always the same. `std::setprecision` only changes how it's displayed.

reformatorsystemu
October 23, 2019 at 2:16 am · Reply

Hello,

I am wondering why my code produces the output as below:

```
    std::cout << std::setprecision(17);

    double small_num = {0.1};

    double mpy_result = 10.0 * small_num;

    double sum_result = small_num + small_num + small_num + small_num + small_num + small_n

    std::cout << mpy_result << '\n';

    std::cout << sum_result;
```

```
1
0.99999999999999989
```

Why the multiplication operation gives different result than addition?

Jose
August 3, 2019 at 10:06 am · Reply

Hi, the conclusion says I should remember three things about floating point numbers but only mentions two...is that a rounding error?

Alex
August 4, 2019 at 5:37 pm · Reply

It's definitely some kind of precision issue. :) Thanks, fixed!

« Older Comments 1 ... 3 4 5