4.8 — Floating point numbers

learncpp.com/cpp-tutorial/floating-point-numbers/

Integers are great for counting whole numbers, but sometimes we need to store *very* large (positive or negative) numbers, or numbers with a fractional component. A **floating point** type variable is a variable that can hold a number with a fractional component, 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.

Tip

When writing floating point numbers in your code, the decimal separator must be a decimal point. If you’re from a country that uses a decimal comma, you’ll need to get used to using a decimal point instead.

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 (created by William Kahan). In this format, a float is 4 bytes, a double is 8 bytes, 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 variables:

```
float fValue;
double dValue;
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.

```
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.

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.

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';

        return 0;
}
```

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	±1.18 x 10^{-38} to ±3.4 x 10^{38} and 0.0	6-9 significant digits, typically 7
8 bytes	±2.23 x 10^{-308} to ±1.80 x 10^{308} and 0.0	15-18 significant digits, typically 16
80-bits (typically uses 12 or 16 bytes)	±3.36 x 10^{-4932} to ±1.18 x 10^{4932} and 0.0	18-21 significant digits
16 bytes	±3.36 x 10^{-4932} to ±1.18 x 10^{4932} and 0.0	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 precision number would require infinite memory to store, and we typically only have 4 or 8 bytes per value. This limited memory means floating point numbers can only store a certain number of significant digits -- any additional significant digits are either lost or represented imprecisely. The number that is actually stored will be close to the desired number, but not exact. We'll show an example of this in the next section.

The **precision** of a floating point type defines how many significant digits it can represent without information loss.

The number of digits of precision a floating point type has depends on both the size (floats have less precision than doubles) and the particular value being stored (some values can be represented more precisely than others).

For example, a `float` has 6 to 9 digits of precision. This means that a float can exactly represent any number with up to 6 significant digits. A number with 7 to 9 significant digits may or may not be represented exactly depending on the specific value. And a number with more than 9 digits of precision will definitely not be represented exactly.

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.

Key insight

A floating point type can only precisely represent a certain number of significant digits. Using a value with more significant digits than the minimum may result in the value being stored inexactly.

Outputting floating point values

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).

We can override the default precision that std::cout shows by using an `output manipulator` function named `std::setprecision()`. **Output manipulators** alter how data is output, and are defined in the *iomanip* header.

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

int main()
{
    std::cout << std::setprecision(17); // show 17 digits of precision
    std::cout << 3.33333333333333333333333333333333333333f <<'\n'; // f suffix means
float
    std::cout << 3.33333333333333333333333333333333333333 << '\n'; // no suffix means
double

    return 0;
}
```

Outputs:

```
3.3333332538604736
3.3333333333333335
```

Because we set the precision to 17 digits using `std::setprecision()`, each of the above numbers is printed with 17 digits. But, as you can see, the numbers certainly aren't precise to 17 digits! And because floats are less precise than doubles, the float has more error.

Tip

Output manipulators (and input manipulators) are sticky -- meaning if you set them, they will remain set.

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

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

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, decimal value 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 <iomanip> // for std::setprecision()
#include <iostream>

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

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

    double d1{ 1.0 };
    std::cout << d1 << '\n';

    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 << '\n';

    return 0;
}
```

```
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 6.6 -- 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 norm. Never assume your floating point numbers are exact.

A corollary of this rule is: be wary of using floating point numbers for financial or currency data.

Related content

For more insight into how floating point numbers are stored in binary, check out the float.exposed tool.
To learn more about floating point numbers and rounding errors, floating-point-gui.de and fabiensanglard.net have approachable guides on the topic.

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). NaN and Inf are only available if the compiler uses a specific format (IEEE 754) for floating point numbers. If another format is used, the following code produces undefined behavior.

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 << '\n';

    double neginf { -5.0 / zero }; // negative infinity
    std::cout << neginf << '\n';

    double nan { zero / zero }; // not a number (mathematically invalid)
    std::cout << nan << '\n';

    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.

Best practice

Avoid division by 0.0 altogether, even if your compiler supports it.

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.