The Pragmatic Programmers

C# Brain Teasers

Exercise Your Mind

Steve Love

edited by Adaobi Obi Tulton

Precision Instruments

Precise.cs

```
var x = 1.1;
var y = 2.2;

Console.WriteLine($"{x} + {y} == {x + y}");
```

Guess the Output

What *precisely* does this program output? Try to guess before moving on.

The program outputs the following:

```
1.1 + 2.2 == 3.3000000000000003
```

Platform Alert

Note that this is one example where .NET Framework produces a different output from .NET because the *default* precision specifier for doubles is different. See Further Reading, on page 6 for references to more information.

Discussion

Floating point arithmetic can produce unexpected results and, as this puzzle demonstrates, it's not an issue that's limited to either very large or very small numbers. Integer arithmetic isn't without its surprises, as we'll see in 9, So, What's Left?, on page ?, but calculations involving floating point numbers are frequently subject to the effects of *rounding*, which is inherent to the way that floating point numbers are stored in memory.

The results of floating point calculations in a program might be unexpected, but are never arbitrary. The reality is that those calculations are governed by strict rules that produce predictable results. However, if you predicted that the output would not be exactly 3.3 but didn't quite guess *precisely* the actual output, don't be too hard on yourself.

Mathematicians have the concept of *real numbers*. These can represent any number, and there are infinitely many of them. Floating point numbers in code look very like real numbers because they're often written using a decimal point (1.23, 3.14159, and so on) just as real numbers are. But floating point numbers aren't the same as real numbers; in the first place there are only a finite number of them.

The floating point types used in C# are represented according to the IEEE Standard for Floating Point Arithmetic, a.k.a. IEEE-754, which makes for very dry reading but specifies exactly how floating point numbers are represented, and how calculations involving them should behave.

The compromise is that a double (a *double precision* number) uses 64 bits, with the result that many numbers don't have an exact representation. Instead, the value stored in memory may be an approximation that's very close to the exact value but differs in its least significant digits.

This compromise is demonstrated in practice in this puzzle's code. The result of adding 1.1 to 2.2 can't be represented exactly in a double so it's been rounded up by a tiny amount to the nearest value that *can* be exactly represented.

However, this explanation still doesn't tell the whole story. The displayed output clearly shows that 3.3 has been rounded, but appears to show that the values for x and y haven't been affected. In fact neither 1.1 nor 2.2 has an exact representation, but the effects of rounding them don't appear when they're written to the screen.

Precision and Tolerance

When a binary floating point number—a double or float value—is written to the screen via Console.WriteLine, the default behavior is to call the value's ToString method to translate the value to a displayable format. In the absence of any further guidance by the programmer, double.ToString automatically rounds the number to its most compact *round-trippable* representation (the behavior in .NET Framework is slightly different; see Further Reading, on page 6 for some links to the Microsoft documentation for more information on that).

For the purposes of this discussion it's sufficient to say that round-trippable means that for some floating point value x, the following is true:

```
double.Parse(x.ToString()).Equals(x);
```

We can provide more detailed instructions on how to represent the value via the format specifier to ToString. For example, the following code uses the G (standing for *General*) format specifier to display the value 1.1 with up to 32 significant digits:

```
Console.WriteLine($"{1.1:G32}");
```

That code produces the following output:

```
1.1000000000000000888178419700125
```

This clearly shows that the double value 1.1 can't be exactly represented in a double, because its *true* value has been rounded up by a tiny amount. However, the most compact *round-trippable* representation of 1.1 is in fact exactly 1.1. Were we to parse the string "1.1" into a double, we'd get *exactly* the same value, including the tiny rounding up. Similarly, the value 2.2 has a round-trippable representation of "2.2", but the most compact representation of the result of adding 1.1 and 2.2 must include the rounding up in order to guarantee that parsing the string back to a double produces the correct value, explaining the output from this puzzle's original code.

Error Magnification

One final puzzle remains. If we print the constant value 3.3 to the console, the output is exactly "3.3", so why is the output of 1.1 + 2.2 so different? The answer to that question is most easily demonstrated in a simple unit test, such as the following:

```
var x = 1.1;
var y = 2.2;

Assert.That(x + y, Is.EqualTo(3.3));
```

Although arithmetically speaking the result of 1.1 + 2.2 is 3.3, this test fails with a result similar to the following:

```
Expected: 3.2999999999999998d
  But was:  3.3000000000000003d
```

The Expected value is the constant number 3.3, but the output shows that its actual internally stored value has been rounded down by a tiny amount because 3.3 can't be represented exactly in a double. Just as importantly, note that the result of 1.1 + 2.2 has been rounded by a similarly small amount, but the rounding has been applied differently. This is because neither of the two values, 1.1 and 2.2, has an exact representation, so they will have been individually rounded before the addition. The result is that the rounding error has been amplified in the result.

It is for this reason that directly comparing floating point numbers for equality in an expression like x == y or x.Equals(y) should usually be avoided; the rounding of floating point numbers that's a direct consequence of how they're stored in memory makes such comparisons unreliable, at best. This behavior is *by design*, as laid out in the IEEE-754 International Standard.

The lesson here is that relying on console (or log file) output to check floating point numerical values *by eye* is unreliable. A unit test is a much better way to diagnose any problems you encounter with floating point arithmetic.

Further Reading

Wikipedia has lots of information on IEEE-754 at

https://en.wikipedia.org/wiki/IEEE_754

The default format specifier for double *values is described in the Microsoft documentation at*

https://learn.microsoft.com/en-us/dotnet/api/system.double.tostring?view=net-8.0#system-double-tostringa

The difference when using the "G" format specifier in .NET and .NET Framework is explained at

https://learn.microsoft.com/en-us/dotnet/standard/base-types/standard-numeric-format-strings#general-format-specifier-g