Subject: FP humor
Posted by mirek on Thu, 05 Aug 2021 12:47:20 GMT
View Forum Message <> Reply to Message

```
CONSOLE_APP_MAIN
{
 x = 1;
 for(int i = 0; i < 10000; i++)
  x *= 0.6;
 DDUMP(Sprintf("%.16g", x));
}
```

Sprintf("%.16g", x) = 4.940656458412465e-324

....and I was thinking that the lowest possible double is 1e-308...

Subject: Re: FP humor
Posted by dolik.rce on Thu, 05 Aug 2021 13:41:12 GMT
View Forum Message <> Reply to Message

Hi Mirek,

mirek wrote on Thu, 05 August 2021 14:47....and I was thinking that the lowest possible double is 1e-308...

You made me do some research and learn something new again :)

To save others the trouble: 1e-308 is smallest normalized value, denormalized values can be around 4.94066e-324. And (not surprisingly) the C++ standard thought about it:
https://en.cppreference.com/w/cpp/types/numeric_limits/denor m_min

Best regards,
Honza

Subject: Re: FP humor
Posted by mirek on Thu, 05 Aug 2021 13:45:20 GMT
View Forum Message <> Reply to Message

dolik.rce wrote on Thu, 05 August 2021 15:41Hi Mirek,

mirek wrote on Thu, 05 August 2021 14:47....and I was thinking that the lowest possible double is 1e-308...

You made me do some research and learn something new again :)

To save others the trouble: 1e-308 is smallest normalized value, denormalized values can be around 4.94066e-324. And (not surprisingly) the C++ standard thought about it: https://en.cppreference.com/w/cpp/types/numeric_limits/denor m_min

Best regards,
Honza

Exactly.

But another interesting issue but perhaps not so suprising fact is that if you multiply variable by 0.6 many many times, it eventually reaches the minimal representable value and then stays at it because of rounding rules...

Subject: Re: FP humor
Posted by mirek on Thu, 19 Aug 2021 15:39:23 GMT
View Forum Message <> Reply to Message

BTW, these are while working on new double<->String conversion routines.

Things are really complicated if you want to do it with absolute possible precision and fast.

E.g. for introduction:

https://www.ryanjuckett.com/printing-floating-point-numbers/

Now I have a nice algorithm tested with 300 billions of samples without an issue. However, I would possibly need help with following problem:

The heart of algorithm is FP with ~60 bit variable mantissa multiplication with 128 bits mantissa constants (this is basically to compute correct pow10), resulting in 128FP number. To prove that there are no input values that would lead to error (which would be less than 2^-70 anyway), I would need to prove that for no variable mantissa bitpattern and no constant, lower 64 bits are never in middle range (half +/- 4), except when they need to be (which is a bit hard to define.. but it is when mantissa has lower bits zero).

Anybody interested in helping with this?
