Why does C print float values after the decimal point different from the input value? [duplicate]

Asked 1 year, 2 months ago Modified 1 year, 2 months ago Viewed 1k times

1

This question already has answers here:

Why IEEE754 single-precision float has only 7 digit precision? (2 answers)

Closed last year.

Why does C print float values after the decimal point different from the input value?

Following is the code.

CODE:

```
#include <stdio.h>
#include<math.h>
void main()
{
    float num=2118850.132000;
    printf("num:%f",num);
}
```

OUTPUT:

```
num:2118850.250000
```

This should have printed *2118850.**132000***, But instead it is changing the digits after the decimal to ***.250000***. Why is it happening so? Also, what can one do to avoid this? Please guide me.

c floating-point precision decimal-point

Share Follow

edited Aug 12, 2021 at 7:34

Evg

23.6k ● 5 ● 40 ● 76

asked Aug 12, 2021 at 7:31

Supriya Bhide

53 ● 1 ● 7

3 Please see Is floating point math broken? and Why Are Floating Point Numbers Inaccurate? – Weather Vane Aug 12, 2021 at 7:33

A `float` can store only about 7 decimal digits significance, which is where yours is diverging. To improve it, use `double`. But think about it. There are only about 2^32 discrete values that a `float` can hold, to represent an infinite range of values, which means that most values *cannot* be exactly represented. Floating point is a trade-off between range and accuracy. – Weather Vane Aug 12, 2021 at 7:34

Additionally see itu.dk/~sestoft/bachelor/IEEE754_article.pdf – Aval Sarri Aug 12, 2021 at 7:35

2 Answers

Sorted by:
Reset to default

Trending (recent votes count more)

0

Your computer uses *binary* floating point internally. Type `float` has 24 bits of precision, which translates to *approximately* 7 decimal digits of precision.

Your number, 2118850.132, has 10 decimal digits of precision. So right away we can see that it probably won't be possible to represent this number exactly as a `float`.

Furthermore, due to the properties of binary numbers, no decimal fraction that ends in 1, 2, 3, 4, 6, 7, 8, or 9 (that is, numbers like 0.1 or 0.2 or 0.132) can be exactly represented in binary. So those numbers are always going to experience some conversion or roundoff error.

When you enter the number 2118850.132 as a `float`, it is converted internally into the binary fraction `1000000101010011000010.01`. That's equivalent to the decimal fraction 2118850.25. So that's why the .132 seems to get converted to 0.25.

As I mentioned, `float` has only 24 bits of precision. You'll notice that `1000000101010011000010.01` is exactly 24 bits long. So we can't, for example, get closer to your original number by using something like `1000000101010011000010.001`, which would be equivalent to 2118850.125, which would be closer to your 2118850.132. No, the next lower 24-bit fraction is `1000000101010011000010.00` which is equivalent to 2118850.00, and the next higher one is `1000000101010011000010.10` which is equivalent to 2118850.50, and both of those are farther away from your 2118850.132. So 2118850.25 is as close as you can get with a `float`.

If you used type `double` you could get closer. Type `double` has 53 bits of precision, which translates to approximately 16 decimal digits. But you still have the problem that .132 ends in 2 and so can never be exactly represented in binary. As type `double`, your number would be represented internally as the binary number `1000000101010011000010.00100001110010101110000001000010` (note 53 bits), which is equivalent to 2118850.132000000216066837310791015625, which is much closer to your 2118850.132, but is still not exact. (Also notice that 2118850.132000000216066837310791015625 begins to diverge from your 2118850.1320000000 after 16 digits.)

So how do you avoid this? At one level, you can't. It's a fundamental limitation of finite-precision floating-point numbers that they cannot represent *all* real numbers with perfect accuracy. Also, the fact that computers typically use binary floating-point internally means that they can almost never represent "exact-looking" decimal fractions like .132 exactly.

There are two things you can do:

1. If you need more than about 7 digits worth of precision, definitely use type `double`, don't try to use type `float`.
2. If you believe your data is accurate to three places past the decimal, print it out using `%.3f`. If you take 2118850.132 as a `double`, and printf it using `%.3f`, you'll get 2118850.132, like you want. (But if you printed it with `%.12f`, you'd get the misleading 2118850.132000000216.)

Share Follow

edited Aug 12, 2021 at 18:53

answered Aug 12, 2021 at 7:43

Steve Summit
40.7k ● 7 ● 65 ● 93

Thank you so much for such a clear and detailed. This cleared all my doubts. Thanks a lot again. – Supriya Bhide Aug 12, 2021 at 9:27

Re "...10 decimal digits of precision. So right away we can see...": This is not a correct inference; some numbers of 10 digits or more are exactly representable in the format commonly used for `float` (IEEE-754 "single", a.k.a. binary32). There is even a number with 105 significant decimal digits that is exactly representable. – Eric Postpischil Aug 12, 2021 at 10:18

@EricPostpischil Answer adjusted. I'd be curious to know that 105-digit number. – Steve Summit Aug 12, 2021 at 18:53

It is 1.401298464324817070923729583289916131280261941876515771757068283889791082685860601486638183836212158203125•10^−45, which is 2^−149. – Eric Postpischil Aug 12, 2021 at 18:58

@EricPostpischil Fascinating. I was thinking that the number of significant decimal digits for an IEEE-754 single-precision float couldn't be more than 24, because each additional fraction bit generally leads to a whole extra decimal digit (which is coincidentally always a 5 :-)). But I was forgetting about what a negative exponent can do, and I think you have sneakily used a denormalized number to bum out a few more digits. :-) (For those following along, it's suggestive that the hex representation of the `float` value Eric's talking about is `0x00000001`. Clearly a very small number.) – Steve Summit Aug 12, 2021 at 19:09

-1

This will work if you use double instead of float:

```
#include <stdio.h>
#include<math.h>
void main()
{
    double num=2118850.132000;
    printf("num:%f",num);
}
```

Share Follow

answered Aug 12, 2021 at 7:40

user16644762

1 Just don't do `printf("num:%.10f",num);` . – Steve Summit Aug 12, 2021 at 8:23

Hi! @SteveSummit I am unable to interpret the meaning of your comment. Can please describe?
– user16644762 Aug 12, 2021 at 8:30

1 @sdgsy4634dfgdfg: Changing `float` to `double` will not result in setting `num` to 2118850.132000. In the format most commonly used for `double`, it will set `num` to 2118850.132000000216066837310791015625. That it looks like 2118850.132000 when printed is an illusion caused by not printing the complete value. That is, OP will still have a floating-point value different from the value of their source numeral, so their problem may just be hidden, not solved. – Eric Postpischil Aug 12, 2021 at 10:14

@sdgsy4634dfgdfg "This will work" seems a little strong. Part of OP's problem is that the precision of type `float` is less than the input number 2118850.132. Now, it's true, type `double` has more precision than that, so switching to `double` will at least help, but it still can't represent 2118850.132 exactly. So switching to `double` and printing with `%.3f` will "work", and as it happens plain `%f` will "work", too, because the default precision is 6. But switching to `double` and then printing with `%.10f` would reveal that the number has still not been represented exactly. – Steve Summit Aug 12, 2021 at 22:52

@SteveSummit So do you think string conversion will be helpful.I think that's the only solution then.
– user16644762 Aug 13, 2021 at 8:44