A Floating Point Problem

by Mark Nelson

Windows Developer's Journal -- May, 1996

Note - this is the unedited original text of the article that appeared in WDJ. The actual published version is somewhat different, but the technical aspects of the article are still accurate.

Probably the scariest thing I have to do at work each day is to walk past Marc Leger's office. Since Marc's office is on the way to the bathroom, I have a tough time avoiding it. So several times a day, I have to face the possibility that Marc will accost me at a time when I am really in a hurry to be elsewhere.

Today was not going to be one of those lucky days when I got off the hook. I slipped down the hallway as quietly as I could, but I wasn't quiet enough. Just as I reached Marc's office, I heard the familiar demand. "Mark! Mark! Come here, I need help!"

Into the breach

With a sigh of sympathy for my bladder, I pulled up a chair behind Marc's desk. He was running Turbo Debugger, and had a program similar to that shown in fp1.cpp.

Marc was working on a customer problem. A Greenleaf Database Library user was attempting to insert floating point data into a database,

and our library was returning an error code indicating a loss of precision. Marc was in his usual state of consternation for two reasons. First, he hadn't seen this problem occur before. Second, he couldn't reproduce the problem when using doubles, just floats.

The smoking gun

The offending code is fairly simple. It converts a C++ double to ASCII representation for storage in a database. After converting and storing the result, the code wants to see if any loss of precision was caused during the conversion process. For example, if I'm using 2 digits after the decimal point, I should get an error if I try to store 3.141. The trailing 1 would be lopped off by the binary to ASCII conversion done by `sprintf()`. Marc's test program from Listing 1 produced the following output:

```
add_double( 1.12 ) returns success
add_double( 1.125 ) returns failure
add_double( d ) returns success
add_double( f ) returns failure
```

As expected, when using two digits of precision, 1.12 is added to the database without error. 1.125 fails, because it will be stored in ASCII as 1.12. Using a double that has been assigned the value of 1.12 works properly. But using a float with a value of 1.12 fails!

Seeing double

The thing that was really bothering Marc was the display on his debugger. He was at the end of the `add_double()` routine, and had all three local variables in his watch window. They all looked perfect. The two floating point numbers had values of 1.12, and the character buffer

contained a "1.12". Even so, the comparison "test == d" had just failed, causing the routine to return an error. Was the compiler somehow at fault? Marc was convinced it was a conspiracy on Borland's part to make his life hell.

Mark's problem was fairly simple, once you looked at it from the right perspective. Numbers such as 1.12 can't be represented exactly in a float, or even in a double. Thus, there are rounding and truncation errors when the conversion takes place. The specific problem in fp1.cpp is the loss of precision found when placing 1.12 in a float. Assigning that value to a double later on doesn't fix the problem, it only perpetuates it.

In the `add_double()` routine, the actual value passed using a float is just a little bit off from the desired value of 1.12. But printing the value out to a buffer using just two digits of precision manages to throw out the error. Thus, when `atof()` converts the buffer back to a double, the error is gone. This means that the input value and the test value are going to differ by a small version.

fp2.cpp is a program that demonstrates how this works. d1 is a double that has been initialized with the value of 1.12. d2 has the same value after being initialized using a float. The output of the program is shown below:

```
f  = 1.12  1.120000005  dump: 29 5c 8f 3f
d1 = 1.12  1.12         dump: ec 51 b8 1e 85 eb f1 3f
d2 = 1.12  1.120000005  dump: 00 00 00 20 85 eb f1 3f
```

As you can see, looking at all three values with just 3 digits of precision makes them look identical. However, when you use a greater precision, the representation error of a float rears its ugly head. The binary dump of the second double shows exactly where the problem lies. The lower

half of the double contains all zeros, a result of having the smaller float value copied into it. Clearly, when comparing (test == d) in fp1.cpp, the compiler will notice the difference.

A Happy Camper

Marc wasn't particularly happy with my explanation. First, it didn't offer an immediate solution to his problem. And second, he resented my glib advice: "Never use floating point numbers if you expect your program to work."

The best answer for our customer would have been to keep all floating point data in doubles at all times. Unfortunately, this solution was impractical. Instead, we created a short routine for her that simply rounded off doubles to 2 significant digits, throwing out anything beyond that. Life would have been much easier if we could have convinced her to simply ignore the error return!

My advice to you is the same: avoid floating point anytime it isn't really necessary. If you do find you need to use it, try to avoid mixing doubles and floats. Stick to the preferred C++ floating format of double for all your data and calculations, and maybe you'll avoid those moments of hysteria that Marc has to constantly deal with.

Return to Mark Nelson's Home Page