

Re: double to int conversion yields strange results

Source: http://coding.derkeiler.com/Archive/C_CPP/comp.lang.c/2005-02/2345.html

From: Tim Rentsch (txr_at_alumnus.caltech.edu)

Date: 02/16/05

Date: 16 Feb 2005 12:39:05 -0800

Bjørn Augestad <boa@metasystems.no> writes:

```
> Below is a program which converts a double to an integer in two
> different ways, giving me two different values for the int. The basic
> expression is 1.0 / (1.0 * 365.0) which should be 365, but one variable
> becomes 364 and the other one becomes 365.
>
> Does anyone have any insight to what the problem is?
>
> Thanks in advance.
> Bjørn
>
> $ cat d.c
> #include <stdio.h>
>
> int main(void)
> {
> double dd, d = 1.0 / 365.0;
> int n, nn;
>
> n = 1.0 / d;
> dd = 1.0 / d;
> nn = dd;
>
> printf("n==%d nn==%d dd==%f\n", n, nn, dd);
> return 0;
> }
>
> $ gcc -Wall -O0 -ansi -pedantic -W -Werror -o d d.c
> $ ./d
> n==364 nn==365 dd==365.000000
>
> $ gcc -v
> Reading specs from /usr/lib/gcc/i386-redhat-linux/3.4.2/specs
> Configured with: ../configure --prefix=/usr --mandir=/usr/share/man
> --infodir=/usr/share/info --enable-shared --enable-threads=posix
> --disable-checking --with-system-zlib --enable-__cxa_atexit
> --disable-libunwind-exceptions --enable-java-awt=gtk
```

comp.lang.c: Re: double to int conversion yields strange results

```
> --host=i386-redhat-linux  
> Thread model: posix  
> gcc version 3.4.2 20041017 (Red Hat 3.4.2-6.fc3)  
> $
```

Having looked at what goes on with this sort of thing a lot during a thread of several months ago, I feel obliged to post a response and try to clear things up a bit.

A couple of postings have tried to sweep the problems under the rug saying things like the problem is inherent in the nature of floating point, or floating point isn't precise, or words to that effect. I think that's both wrong and misleading. First, floating point is **precise**; what it is not is **exact**. Floating point does behave according to precise rules, which are implementation specific to some extent, but they are precise nonetheless. The rules often don't give the results we expect, because in mathematics we expect the results to be exact, which floating point calculations are not. In spite of that, floating point calculations do behave precisely, and if we learn how they behave we can use them with confidence.

Furthermore, the question raised is not about the precision of the answer but about whether floating point calculation is deterministic. If you do the same calculation twice, do you get the same result? And that question wasn't really addressed in other responses.

At least one posting attributed the problem to the x86 platform and problems with gcc, with correcting/contradicting posts following. The contradiction is both right and wrong. The behavior of the program listed above is conformant with the C standard. But, this kind of behavior does tend to show up erroneously on the x86 platform, and gcc is known to have bugs around behavior of floating point (eg, 'double') operands w.r.t. conforming to the C standard, especially in the presence of optimization. It's probably worth bearing that in mind when testing for how a program "should" behave. (See also the workaround below.)

The key here, as was pointed out, is that the calculation '1.0 / d' in the assignments

```
n = 1.0 / d;  
dd = 1.0 / d;
```

is done (on the x86) in extended precision. In the second assignment, the extended precision value is converted to 'double' before doing the assignment; but in the first assignment, the extended precision value is **not** converted to 'double' before being converted to 'int'. Thus, in the (as was also posted) corrected code

```
n = (double) (1.0 / d);  
dd = 1.0 / d;
```

comp.lang.c: Re: double to int conversion yields strange results

we could reasonably expect that an assertion

```
assert( (int) dd == n );
```

to succeed. (Note: "reasonably expect" but not "certainly expect"; more below.)

It is the conversion to 'double' before the conversion to 'int' that makes the two assigned expressions alike. The C standard requires this; see 6.3.1.4, 6.3.1.5, and 6.3.1.8. In particular, either casting with '(double)' or assigning to a 'double' variable is required to convert an extended precision value to a value with exactly 'double' precision. That's why the corrected code behaves more as we expect.

That an assignment to a double variable sometimes requires a conversion can lead to some unexpected results. For example, consider

```
#define A_EXPRESSION (*some side effect free expression*)
#define B_EXPRESSION (*some side effect free expression*)

double a = A_EXPRESSION;
double b = B_EXPRESSION;
double c = a + b;
double d = (A_EXPRESSION) + (B_EXPRESSION);
```

Normally we might expect that 'c' and 'd' can be used interchangeably (when doing common subexpression elimination inside the compiler, for example), but the conversion rules for C make this not so, or at least not always so. Using temporaries in calculations using 'float' or 'double' changes the semantics in a way that doesn't happen with 'int' values.

To return to the original question, how is it that the value of 'n' differs from '(int) dd'? The behavior of the conversion to 'int' is required by the standard to truncate. If the conversion from extended precision to 'double' also truncated, there would be no discrepancy in the program above. But the conversion (in this implementation) from extended precision to 'double' doesn't truncate, it rounds. This behavior conforms to the C standard, which requires that the result be exact when possible, or either of the two nearest values otherwise (assuming that there are such values represented in 'double'). Which of the nearest values is chosen in the latter case is "implementation defined".

So, as far as I can tell, an implementation could convert from extended precision to 'double' by doing "statistical rounding" — that is, a mode in which rounding is done non-deterministically — and still be conformant. I don't know of any hardware that actually does this, but as far as I can tell non-deterministic rounding is allowed.

comp.lang.c: Re: double to int conversion yields strange results

Bottom line is, putting in the '(double)' cast will very likely make the code deterministically yield 'n == (int) dd', but the standard doesn't guarantee that it will. It may be possible to guarantee deterministic behavior by setting the "rounding direction mode", or by setting the "dynamic rounding precision mode", if supported; please see Annex F.

=====

Incidental note on getting around problems with gcc. Because gcc on the x86 platform sometimes fails to convert extended precision values to 'double' precision when the standard semantics require it, it's nice to have a way to force a '(double)' conversion to take place.

The inline function

```
inline double
guarantee_double( double x ){
    return * (volatile double *) &x;
}
```

is a pretty solid way of doing that.