cilu January 5th, 2005 03:56 AM

C++ General: How is floating point representated?

Q: What is IEEE 754 standard?

A: IEEE Standard 754 floating point is the most common representation today for real numbers on computers, including Intel-based PC's, Macintoshes, and most Unix platforms.

Q: Is this the format use by Microsoft VC++ also?

A: Microsoft Visual C++ is consistent with the IEEE numeric standards. There are three internal varieties of real numbers. Real*4 and real*8 are used in Visual C++. Real*4 is declared using the word **float**. Real*8 is declared using the word **double**. In Windows 32-bit programming, the **long double** data type maps to **double**. There is, however, assembly language support for computations using the real*10 data type.

Q: What is the format specified by the standard?

A: IEEE floating point numbers have three basic components: the sign, the exponent, and the mantissa. The sign bit is 0 for positive, 1 for negative. The exponent's base is two. The exponent field contains 127 plus the true exponent for single-precision, or 1023 plus the true exponent for double precision. The first bit of the mantissa is typically assumed to be 1.f, where f is the field of fraction bits.
To learn more about the standard see:

- IEEE Standard 754 Floating Point Numbers
- IEEE Standards Site

Q: What is the range of real numbers in VC++?

A:

float (4 bytes) : 1.175494351E-38 to 3.402823466E+38, significant decimal digits: 6
double (8 bytes) : 2.2250738585072014E-308 to 1.7976931348623158E+308, significant decimal digits: 15
real*10 (10 bytes) : 3.37E-4932 to 1.18E+4932, significant decimal digits: 19

Q: I have a problem with the following code.

Code:

```
int main()
{
  float a = 2.501f;
  a *= 1.5134f;
  if (a == 3.7850134) cout << "Expected value" << endl;
  else cout << "Unexpected value" << endl;
}
```

Why does the program outputs "Unexpected value", because 2.501 * 1.5134 = 3.7850134?

A: Floating-point decimal values generally do not have an exact binary representation. This is a side effect of how the CPU represents floating point data. Different compilers and CPU architectures store temporary results at different precisions, so results will differ depending on the details of your environment. If you do a calculation and then compare the results against some expected value it is highly unlikely that you will get exactly the result you intended.

To summarize, never make such a comparison:

Code:

```
if (a == b) ...
```

Instead make sure that the result is greater or less than what is needed, with a given error.

Code:

```
if( fabs(a - b) < error) ...
```

The above example should be rewritten like this:

Code:

```
#define EPSILON 0.0001  // Define your own tolerance
#define FLOAT_EQ(x,v) (((v - EPSILON) < x) && (x <( v + EPSILON)))
int main()
{
  float a = 2.501f;
  a *= 1.5134f;
  if (FLOAT_EQ(a, 3.7850)) cout << "Expected value" << endl;
  else cout << "Unexpected value" << endl;
}
```

However, since float has 6 significant decimals you might want to have an EPSILON value not grater than 0.000001. It depends on the tolerance you need. But you cannot use an EPSILON of 0.0000001 because it that case it exceeds the float precision.

In order to avoid any misleading you should understand that there can be only 6 decimal digits in the result. But this does not imply 0.000001! When dealing with all small values you could just as well have an epsilon of 0.0000000000000001 providing the values compared to are equaly small enough. In the case of a float value of 12345.6789, the float is only reliably correct to the first 6 decimal digits, so, it's at best guaranteed accurate only to 0.1. Using the epsilon macro to an accuracy of 0.0001 may not actually help in establishing equality.

It is a common misconception that *epsilon* when dealing with floats is (or can be) an absolute value. It is not! Epsilon (as in the FLT_EPSILON or DBL_EPSILON definitions) is the minimal representable value, but in order to apply it to a result, you have to scale epsilon to the same exponent as the values you are comparing.

Code:

```
// float.h
#define DBL_EPSILON    2.2204460492503131e-016 /* smallest such that 1.0
+DBL_EPSILON != 1.0 */
#define FLT_EPSILON    1.192092896e-07F        /* smallest such that 1.0
+FLT_EPSILON != 1.0 */
```

Using the real FLT_EPSILON or DBL_EPSILON definition (scaled to match operands) in order to compare for equality is possible, but requires a considerable amount of code. The FLOAT_EQ() macro is a good easy alternative, but you do need to be aware that it can only perform it's job properly when the values tested are within the accuracy range of the type used (float/double).

Code:

```
float a = 51234.1f;
a*= 79.6787f;

if (FLOAT_EQ(a,4082266.48367)) ...
```

Each of the floats used (551234.1f and 79.6787f) are suffienctly accurate (either float is only 6 decimal digits). The resulting float however has 12 decimal digits. Even though you have attempted to adjust for the inequality with the FLOAT_EQ() macro, it still returns false. In fact, the above returns false up to the point where we set EPSILON to 1.0!

Q: Why this inaccuracy of floating type representations and not of integer types also?

A: An integer type number is a string of bits that represent the powers of two, and these powers sum to give the decimal number. For instance 1011 is in decimal 8 + 2 + 1, which is 11.

On the other hand a floating type number is a string of bits that represent the inverted powers of two. For instance 0.1011 is decimal 1/2 + 1/8 + 1/16, which is 0.6875. While you can accurately represent some decimal values (like 0.5, 0.25, 0.75, 0.625,...) you can't accurately represent all decimals values (like 0.1).

Q: The following program outputs "Expected value" both in Release and Debug builds. Why?

Code:

```
int main()
{
  float a = 2;
  a *= 1.5;
  if (a == 3) cout << "Expected value" << endl;
  else cout << "Unexpected value" << endl;
}
```

A: That is because 1.5 has and exact representation in binary: 2^0 + 2^-1, which is 1.1 binary and when you multiply it by 2, increasing the exponent by 1, it yelds an exact value of 3.0 without any rounding.

Q: But the next program outputs "Expected value" in the Debug build and "Unexpected value" in the Release and I don't know why.

Code:

```
int main()
{
  float a = 0.1;
  a*=10;
  if (a == 1.0) cout << "Expected value" << endl;
  else cout << "Unexpected value" << endl;

  return 0;
}
```

A: In Debug build, the value will get actually stored into the stack variable *a* before comparison. This conversion from the FPU stack to float *corrects* the floating point error because of rounding. In a Release build, the variable gets optimised away, and the value on the FPU stack is compared to 1.0 (no conversion to float happens).

Q: Where from can I learn more about floating point comparison?

A: See the Comparing floating point numbers article by Bruce Dawson.

Credits: This FAQ was written with the help of OReubens
<br><br>

cilu

January 16th, 2005 08:14 AM

Re: C++ General: About floating point representation

Q: When should I use float and when double?

A: There is a short answer and a long answer.

The short answer is that if precision is less of a concern than storage, consider using type float for floating-point variables. Conversely, if precision is the most important criterion, use type double.

The long answer is a little bit detailed. Floating point types are very good for mathematical/statistical type applications where you need a high amount of precision but you do not need decimal accuracy. For float you will have a result that is only reliably correct to 6 decimal positions. Double will have a result correct to 15 decimal positions.

For bussiness software, floats/doubles are often a bad choice because of their inherent inaccuracy when dealing with decimal digits. In fact, some countries prohibit by law the use of accounting software that uses floating point variables. Many specifications for bussiness software also prohibit the use of floating point. The solution in these cases is then to either use regular **int** factored by a power of 10 (ex: 1.23 would be stored as 123 with an implicit/explicit factor 100). Or use a decimal BCD (Binary Coded Decimal) type storage (the way Cobol stores its values). C/C++ have support for integers (the implicit/explicit) factoring you will have to provide yourself. There is no built in support for BCD.

With C++ however it is fairly simple to create a class that will handle (large) decimal numbers without using floating point. Even if the specifications for bussiness software do not specify explicitely that you can/can't use floating point, you might consider not using them. It could save you a headaches and tracking down of weird problems dealing with floating point rounding errors.

Q: So, I should not use float/double for bussiness software?

A: For bussiness software, floats will never do, simply because you can only represent very *simple* values accurately (6 decimal digits is not a lot). You will always want to use double because of the higher decimal accuracy of 15 decimal digits. And even with the added precision and range of doubles, they are still quite often a major source of problems when dealing with the need for decimal accuracy.

FPU math is always only an approximation, and that all results are *off* by a *small* amount. For mathematical/statistical type problems, 3D calculations and the like, the inaccuracy is often not a real (no pun intended) problem, and if it is, you just use a higher precision type to get more accurate (but still off) results. There are compilers/libraries that offer even more accuracy than real*10 does.

But bussiness software is a different ballgame. Even an inaccuracy of only 1 cent is a huge problem in accounting or banks. And even a real*1000 could still cause a 1 cent rounding problem (as strange as this may seem).

BCD or factored int's do not have these particular problems, they have a few problems of their own, but they are much more easily defined because they reflect the way we learned how to do decimal math in school. It's this that makes them better suited for bussiness software.

Using floating point seems so deceptively easy that you often won't reconsider anything else. And by the time you're in the spot where you have to deal with floating point rounding/conversion problems, you're usually too far ahead to scrap them all in favour of BCD/factored int's.
<br><br>

All times are GMT -5. The time now is 11:19 AM.