Cross-Platform Issues With Floating-Point Arithmetics in C++

Günter Obiltschnig, Applied Informatics Software Engineering GmbH
guenter.obiltschnig@appinf.com

ACCU Conference 2006

Abstract. The C++ standard does not specify a binary representation for the floating-point types float, double and long double. Although not required by the standard, the implementation of floating point arithmetic used by most C++ compilers conforms to a standard, IEEE 754-1985, at least for types float and double. This is directly related to the fact that the floating point units of modern CPUs also support this standard. The IEEE 754 standard specifies the binary format for floating point numbers, as well as the semantics for floating point operations. Nevertheless, the degree to which the various compilers implement all the features of IEEE 754 varies. This creates various pitfalls for anyone writing portable floating-point code in C++. These issues, and ways how to work around them, are the topic of this paper.

1 Introduction

As with integer types, the C++ standard does not specify a storage size and binary representation for floating-point types. The standard specifies three floating point types: `float`, `double` and `long double`. Type `double` must provide at least as much precision as `float`, and `long double` must provide at least as much precision as double. Furthermore, the set of values of the type `float` must be a subset of the values of type `double` and the set of values of the type `double` must be a subset of the values of type `long double`.

2 IEEE 754 Floating Point Representation

Although not required by the standard, the implementation of floating point arithmetic used by most C++ compilers conforms to a standard, IEEE 754-1985 [1], at least for types `float` and `double`[1]. This is directly related to the fact that the floating point units of modern CPUs also support this standard. Exceptions are VAX and certain Cray machines, which use proprietary formats. The IEEE 754 standard specifies the binary format for floating point numbers, as well as the semantics for floating point operations. The standard is currently (Spring 2006) undergoing a revision.

Byte order also applies to floating point types, so on a big-endian machine, the constant $\pi = 3.1415926...$ (or, strictly speaking, an approximation of it) as a `float` value, will be stored

[1] On some compilers, support for IEEE 754 floating point arithmetic must be explicitly turned on with a compiler option.

as `0x40 0x49 0x0F 0xDB` (starting from the lowest address), while on a little-endian machine it will be stored as `0xDB 0x0F 0x49 0x40`. Of all three floating point types, `long double` is the least portable. Some systems do not support `long double` directly; on these systems, it is the same size as a `double`. Also, on Linux IA32 platforms, `long double` uses the native Intel 80-bit floating point format. On most Unix platforms, `long double` is 128-bit, often implemented in software. So, byte order issues aside, `float` and `double` values usually can be exchanged in binary format between platforms; `long double` values can not. Table 1 gives an overview of the three floating point types.

C++ Type	**Precision**	**Size in bits**	**Fraction bits**	**Exponent bits**
`float`	single	32	23	8
`double`	double	64	52	11
`long double`	quadruple/ double extended	128/80	112/64	15

Table 1: Floating point types and their sizes (sign bit not shown)

An IEEE floating-point number consists of three parts: the sign bit, the mantissa (stored as fraction) and the exponent. So, the actual value of a floating point number is

$$Value = Sign \cdot Mantissa \cdot 10_2^{Exponent}$$

Notice the difference between mantissa and fraction. An IEEE floating point number is normalized such that the most significant bit of the mantissa is always one, thus it is not necessary to store it (an exception to this are the so-called denormal or subnormal numbers). This means that for float values, the mantissa is actually 24 bits. For example, the binary floating point number $1.11101000010111111000111 * 10^{-01111110}$ (which, in decimal, is $2.24 * 10^{-38}$) will be stored as

Sign	**Exponent**	**Fraction**
0	00000001	11101000010111111000111

In this example, although the binary exponent is –01111110, which is –126 in decimal, it is stored as 00000001. The reason for this is that the exponent is not encoded in two's complement, as one would expect. A fixed offset, called a bias, is added to the exponent, with the result that the biased exponent is always a positive integer. For single precision floating point numbers, the bias is 127; for double precision numbers it is 1023. Figure 1-3 shows the storage layout of IEEE floating point numbers for 32-bit single and 64-bit double precision values.

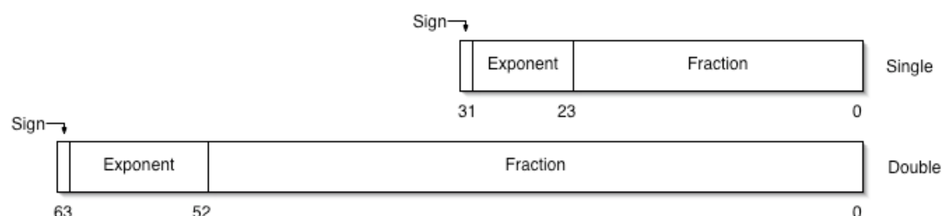

Figure 1: Storage layout for IEEE single precision and double precision floating point values

On some platforms, most notably IA32, the floating point hardware supports only one format (in case of IA32, this is the 80-bit double extended format). Prior to an arithmetic operation, all operands in a different format must be converted to the native format, and eventually the

result must be converted back. This gives the opportunity for a whole new class of nasty surprises, caused by the loss of precision when intermediate results are stored in memory (and thus need to be rounded) for later use.

3 IEEE 754 Special Features

The IEEE 754 standard has some special features, and they are a major reason for problems when porting floating point code between platforms. The features are discussed in the following sections.

3.1 NaN (Not a Number)

IEEE 754 requires that computations continue, even in case of an exceptional condition, such as dividing by zero or taking the square root of a negative number. The result of taking the square root of a negative number is a *NaN* (Not a Number). NaN is represented by a bit pattern that does not yield a valid number. The exponent is all ones and the fraction is non-zero. There are actually two flavors of NaN. *QNaN* (quiet NaN) has the most significant fraction bit set and is the result of an operation when the result is not mathematically defined (indeterminate). *SNaN* (signaling NaN) has the most significant fraction bit cleared and is used to signal an exceptional condition or invalid operation. Additionally, a NaN can be positive or negative. Some implementations do not distinguish between QNaN and SNaN. Furthermore, some implementations always generate negative NaNs, independent of the sign of the operands. Since the bit pattern for the fraction part of a NaN is not exactly specified (other than that it must be non-zero, and for the most significant bit), there is a whole family of NaN values. Implementations are free to use the fraction bits for whatever they like.

There is no straightforward portable way to check whether a given value is NaN. Some platforms, as part of the C library, provide the functions `isnan()` and `isnanf()`, but these are not standardized. The C++ standard library specifies a class `numeric_limits<>`, with specializations for `float`, `double` and `long double`. These classes define the static methods `quiet_NaN()`, `signaling_NaN()`, `has_quiet_NaN()` and `has_signaling_NaN()`, but these cannot be used for that purpose. Since there is no single value for NaN, you cannot just compare a value with the result of `quiet_NaN()` for equality. The result of every arithmetic operation involving at least one NaN is again a NaN. The comparison of a NaN with another value, including another NaN, always yields false. So, one possible workaround would be to check if a value is not equal to itself (`x != x`), which, in case of a NaN, yields true.

The result of a conversion of a NaN to a string, necessary for output to the console, is not standardized and varies between platforms. Similarly, converting a NaN to an integer yields an undefined result.

3.2 Infinity

The result of a divide by zero is *infinity*. An exception is 0/0, which yields NaN. Similar to NaN, infinity is represented by a bit pattern that does not stand for an ordinary number. The exponent is all ones and the fraction is zero. Infinity can be positive or negative. Testing for positive or negative infinity is straightforward, as the `numeric_limits<>` class provides the static method `infinity()`, which returns the value representing positive infinity, which you can use for comparison.

As with NaN, the result of a conversion of infinity to a string is not standardized and varies between platforms. The result of converting infinity to an integer is undefined as well. Table 2 shows the results of various operations involving infinity.

Operation	Result
x / ±Infinity	0
±Infinity * ±Infinity	±Infinity
x / 0 (for x != 0)	±Infinity
±0 / ±0	NaN
Infinity + Infinity	Infinity
Infinity - Infinity	NaN
±Infinity / ±Infinity	NaN
±Infinity * 0	NaN

Table 2: Infinity Operations

3.3 Signed Zero

Zero is represented by a bit pattern, in which both the exponent and the fraction are all zeros. There is still the sign bit, so zero can be positive or negative. Although +0 and –0 are distinct numbers, they compare as equal.

3.4 Gradual underflow and denormal numbers

If the result of an operation lies between zero and the smallest number that can be represented by a normalized floating point value, you are in trouble. You know that the value is not zero, nevertheless you are forced to treat it as zero. This leads to the undesirable situation that `x – y == 0` for `x != y`. In a longer operation, this can lead to a significant error, or even to an unexpected divide by zero. To account for these cases, IEEE 754 defines a concept called *gradual underflow*. If a number has a zero exponent, its exponent is not required to be normalized and the most significant bit of the exponent is considered zero (remember that this bit is not actually stored). Such denormal (or subnormal) values have a lower precision than normalized numbers, but this is still better than treating them as zero.

The smaller a denormal number, the smaller will be its precision. Eventually, a denormal number will degenerate to zero. Denormalized numbers are essential for guaranteeing that if two numbers are different, then the result of subtracting one from the other is not zero. To make things more clear, here is an example that shows how denormal numbers work. Say you have two 32-bit floating point values, $8.97*10^{-38}$ and $8.95*10^{-38}$. When you subtract the second number from the first, the result is $2.0*10^{-40}$ (or $1.99999*10^{-40}$), a number that cannot be represented by a normalized 32 bit floating point value. Table 3 shows how these values are represented internally and how those representations must be interpreted. You can see that the result is a denormal number, because its exponent is zero.

Decimal	Internal	Interpret as (binary)
$8.97*10^{-38}$	0 00000011 11101000010111111000111	1.11101000010111111000111*10^11
$8.95*10^{-38}$	0 00000011 11100111010010001100110	1.11100111010010001100110*10^11
$1.99999*10^{-40}$	0 00000000 00000100010110110000100	0.00000100010110110000100*10^1

Table 3: Representation of denormal numbers

Denormal numbers are hard to implement, especially in hardware, so not all systems implement them. This is one of the reasons why the same calculation can lead to different results on different systems even if both systems use IEEE arithmetic.

3.5 Rounding modes, exceptions and flags

Rounding occurs whenever the result of an operation cannot be exactly represented as a floating point number. IEEE 745 specifies four rounding modes:

- rounding towards nearest,
- rounding towards zero,
- rounding towards positive infinity, and
- rounding towards negative infinity.

In C++, there is no standardized interface to set or query the current rounding mode, and many platforms have no such interface at all at library level.

Whenever an exceptional condition occurs during a computation, the default behavior is to deliver a special result (NaN, infinity) and continue with the computation. This may not always be the best alternative, so the IEEE standard also specifies a trap mechanism, where an exception will lead to the invocation of a trap handler. Additionally, exceptions are also reported by setting the values global status flags accordingly. The standard defines five exception classes:

- overflow,
- underflow,
- division by zero,
- invalid operation, and
- inexact.

However, in C++ again there is no standardized way to set trap handlers or test the status flags. In contrast, the C99 standard defines functions for working with rounding modes, exceptions and flags (called the floating point environment).

4 Issues not addressed by IEEE 754

Neither the IEEE 754 nor the C++ standard specify how transcendental functions like `sin()`, `cos()`, `exp()`, etc. have to be implemented. Therefore, for the same arguments, the results that these functions return on various platforms might not be exactly the same. This is another reason for getting non-identical results for the same expression on different platforms. The standard also does not specify how the conversion from decimal floating point values to their binary representation and vice-versa must be implemented. Especially the string representations of NaN and infinity differ between platforms. Be also cautious when using the `pow()` function. On some systems, it will only work for positive and integral exponents. Instead of `z = pow(x, y)`, use the following when an exponent is negative or non-integral:

```
z = exp(y * log(x))
```

5 IEEE 754 API Support

IEEE 754 does not specify an API for floating-point operations. Therefore, writing cross-platform floating-point code that relies on the special IEEE 754 features can become problematic. Table 4 gives an overview of the API provided by different platforms.

	C99	Windows	Solaris
Headers	<fenv.h> <math.h>	<float.h> <math.h>	<ieeefp.h> <math.h>
Rounding Modes			
downward	FE_DOWNWARD	RC_DOWN	FP_RM
upward	FE_UPWARD	RC_UP	FP_RP
to nearest	FE_TONEAREST	RC_NEAR	FP_RN
toward zero	FE_TOWARDZERO	RC_CHOP	FP_RZ
Flags			
divide by zero	FE_DIVBYZERO	SW_ZERODIVIDE	FP_X_DZ
inexact	FE_INEXACT	SW_INEXACT	FP_X_IMP
overflow	FE_OVERFLOW	SW_OVERFLOW	FP_X_OFL
underflow	FE_UNDERFLOW	SW_UNDERFLOW	FP_X_UFL
invalid	FE_INVALID	SW_INVALID	FP_X_INV
Operations			
clear flags	feclearexcept()	_clearfp()	fpsetsticky()
check flag	fetestexcept()	_statusfp()	fpgetsticky()
set rounding mode	fesetround()	_controlfp()	fpsetround()
get rounding mode	fegetround()	_controlfp()	fpgetround()
test for infinite	isinf()	_finite()	fpclass()
test for NaN	isnan()	_isnan()	isnanf() isnan()
copy sign	copysignf() copysign()	_copysign()	copysign()
save environment	fegetenv()	_controlfp()	fpsetround() fpsetmask()
restore environment	fesetenv()	_controlfp()	fpgetround() fpgetmask()

Table 4: Comparison of floating point APIs

6 Conclusions

Although most modern platforms support IEEE 754 floating point arithmetics, writing cross-platform floating point code is not easy. In this paper, the issues that must be taken into account when writing portable floating point code have been discussed and the APIs provided by different platforms have been compared. To summarize,

- ordinary float and double values can be exchanged in binary form between systems that implement IEEE 754, long double values cannot;
- NaN and denormals are problematic;
- byte order must be taken into account;
- the same code may produce slightly different results on different systems;
- you have to implement your own portable API for working with some IEEE 754 features.

References

[1] Kahan, W., “Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic”, University of California Berkeley, 1996.