Base One International Corporation
44 East 12th Street
New York, NY 10003
212-673-2544
info@boic.com
www.boic.com

High-Precision C++ Arithmetic

- Base One's Number Class fixes the loopholes in C++ high-precision arithmetic

Base One's Number Class is a general-purpose alternative to the built-in numeric data types of C++, with added safeguards against a variety of common (and not-so-common) errors. The Base/1 Number Class is built on an innovative technology for performing highly precise mathematical calculations. (U.S. Patent Number 6,384,748)

The Number Class lets programmers eliminate unacceptable rounding errors and destructive, high-order truncation. Programmers can easily replace numeric datatype declarations (double, int, unsigned long, etc.) and gain efficient, high-precision arithmetic on numbers and fractions with up to 100 decimal digits.

While C++ has become the tool of choice for building all types of applications, it still suffers from limitations that can be traced to its roots as a scientific programming language for minicomputers. For example, C++ lacks features supporting high-precision decimal arithmetic, which have long existed on mainframe computers because of their orientation toward large-scale business applications.

Although C/C++'s native numeric data types work fine in many situations, their underlying binary representation and limited precision present problems that become more apparent as the number of significant digits increases. Thus, for example, native C++ numbers lose accuracy because of rounding, or worse, results can be grossly in error because of failure to detect the truncation of high-order digits in a calculation that produces an overflow. While these shortcomings of C/C++ can be overcome with extra programming effort, one must be diligent, and it's easy to make subtle mistakes that don't show up for years.

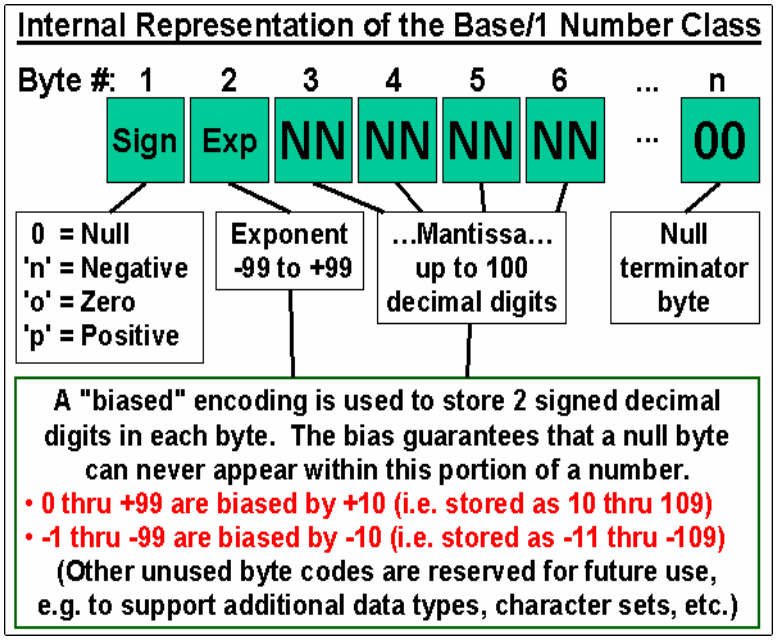
But the beauty of C++ is that it allows you to solve a problem once and reuse that solution from then on. Programmers can make use of Base One's Number Class just as easily as the native C++ number types. Existing applications are easy to convert. With a few localized changes to include the required header files, huge applications can be made to support greatly enhanced precision simply by recompiling.

Because it addresses such a fundamental and general language feature, the Base/1 Number Class applies to a wide range of application types:

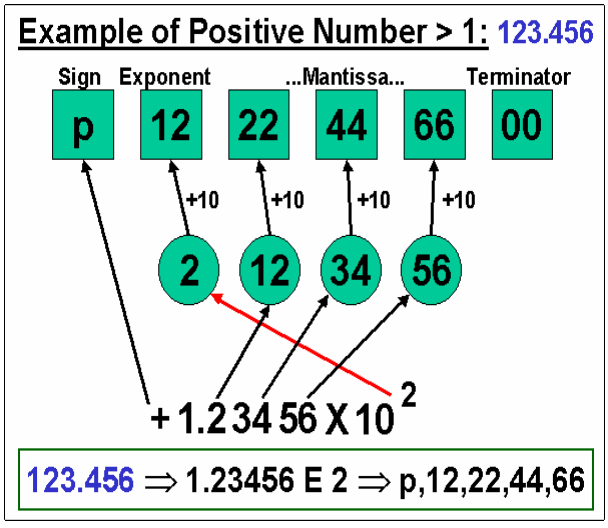
- financial applications: e.g. banking, securities, insurance, and accounting
- applied mathematics: engineering, physical sciences, etc.
- pure mathematics and computer science

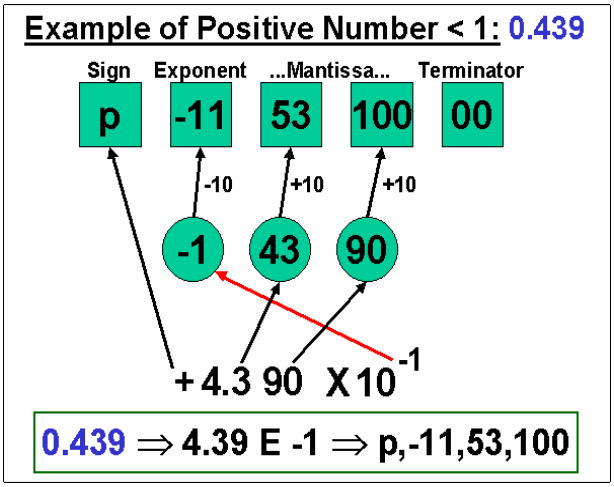
Advantages Over C++ Built-In Numeric Types

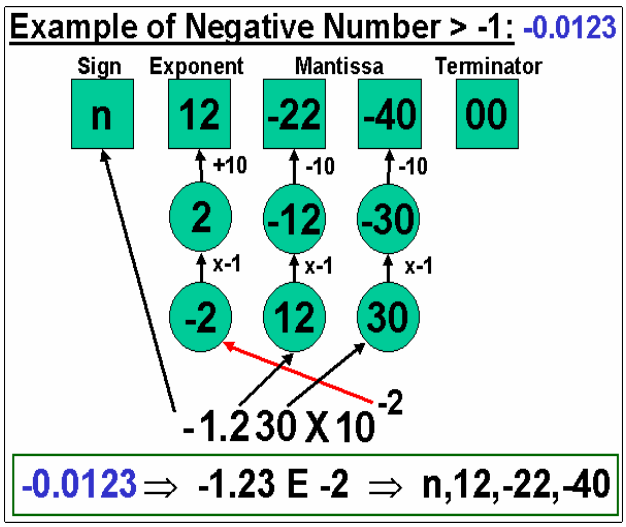
The Number Class provides numbers that can have guaranteed precision, i.e. length or number of significant figures. Internally, the Number Class uses algorithms have some similarities to BCD (Binary Coded Decimal) to do its arithmetic, and a number is stored as a compacted, variable length string. Externally, the Number Class behaves like a floating point number, except that it's better behaved. There's no pervasive, destructive rounding going on. Arithmetic is done exactly.

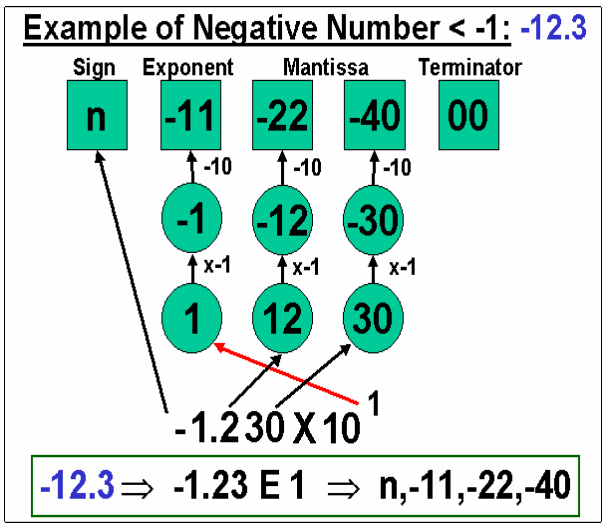


The Number Class efficiently supports numbers that are much larger than can be held by any C++ numeric data type. The Number Class is designed to hold positive and negative numbers ranging from the very small, 1 x 10**-99, to the very large, 9.999... x 10**99. The maximum precision supported is 100 (length) and a maximum scale of 100.









The Number Class uses a variable-length representation that allows numbers to be manipulated as raw null-terminated character strings, which are supported very efficiently in C++. In performing the encoding, the first step always is to canonicalize the number into an equivalent form that has a mantissa with a single non-zero digit to the the left of the decimal point. Digits are then processed in a pairwise fashion to pack them into each byte, with the addition of a single trailing zero if needed to fill the last byte. Sign reversals are performed in such a way as to guarantee the collatability of signed numbers and fractions. Finally, a bias of plus or minus 10 adjusts the byte values to assure that a null byte can never occur in either the exponent or the mantissa.

Decimal floating point representation avoids repeating binary fractions:

It is commonly known that some simple fractions, like 1/3, can not be exactly represented in decimal notation. Depending on the choice of base, there will always be certain fractions that can only be represented by an infinitely repeating series of digits to the right of the decimal point. Because C++ number types use an underlying binary representation, most decimal fractions have no exact representation as standard C++ floating point numbers. For example even the simple decimal number 0.1 (one tenth) translates into the repeating binary fraction, .000110011001100..., ad infinitum. This means that C++ floating point is incapable of exactly representing most decimal fractions, regardless of the choice of precision.

On the other hand, the Number Class uses a compact decimal encoding, so it can exactly represent any decimal fraction up to 100 digits of precision. Since people are accustomed to dealing with decimal numbers, this is the form in which numbers usually originate, and also the form in which results are reported. Financial applications are a classic example of the need for exact decimal arithmetic, because large accounts may be required to balance right down to the penny. By using the Number Class, you eliminate the fundamental inaccuracy of C++'s binary number types.

Much higher precision limits:

The most important general determinant of accuracy is the number of significant digits a number can represent, regardless of whether a binary or decimal encoding is used. For C++ numbers on a PC, the limit comes down to a 64 bit maximum that standard C++ number types can occupy. This translates into a maximum of 19 significant decimal digits for fixed point C++ numbers, or about 16 significant digits for floating point. (Remember, the C language evolved from minicomputers.) The Number Class supports up to 100 significant decimal digits, so it can be vastly more accurate than any C++ number. This makes our Number Class a great tool for scientific and mathematical applications that demand extreme precision.

Number Class uses a fixed scale instead of a floating decimal point:

Scientific applications can be satisfied with a floating decimal point, so long as a fixed precision can be guaranteed. This is NOT a sufficient form of precision for business and database applications. If the decimal point is allowed to float, then, for example, you can't depend on the accuracy to a particular number of digits to the right of the decimal point. In other words, "accurate to the penny" means knowing that no operation caused the value to the right of the decimal point to be lost through rounding. For this reason Number Class uses a fixed scale that can satisfy both business and scientific applications.

Space-efficient variable length encoding stores only the actual digits required:

On a PC, for example, C++ supports native fixed and floating point numeric types that occupy up to 8 bytes (64 bits). Floating point is able to represent a much wider range from tiny fractions to very large numbers, but the tradeoff is a reduction in precision. In either case, C++ numbers use a fixed-length encoding, so that a variable of a declared number type will always be the same size, which must be large enough to hold the largest possible value this type can represent. The Number Class, however, is a variable-length numeric encoding built upon standard C++ null-terminated string classes. Only the amount of space actually needed to hold the current value is allocated. A single null byte is all that is required to store the NULL value, and a simple decimal number can be stored in as little as two bytes, while a number with as many as 100 significant decimal digits can be stored in another variable, yet these are both instances of the same ubiquitous Number Class type. For a typical distribution of decimal number values in a real application, the Number Class' variable length can yield dramatic reductions in the consumption of both memory and disk storage.

The Number Class allocates and stores only the digits that are required to exactly represent the particular value it holds, without needless padding. When there is a mix of long and short numbers, as is typical in many applications, use of Base One's Number Class substantially reduces memory and disk space requirements.

No matter how may digits of precision you sometimes need, the number is stored in the minimum amount of space that the specific number requires. Null (blank) numbers require one byte. Zero requires 2 bytes. A non-zero, non-null number always requires at least 4 bytes. One or two digits needs 4 bytes; 3 or 4 digits, 5 bytes; 5 or 6 digits, 6 bytes, etc.. The formula is:

(3 + ((NumDigits + 1) / 2))

with the division performed being an INTEGER division (i.e. rounded down). (No extra space is required to store either the number's length or its decimal point in the Number Class.)

As the precision (length) of numbers is increased, the fixed overhead of the sign, exponent, and terminator bytes diminishes in importance, and the size approaches 2 decimal digits per byte. Note that this format does NOT need to store zeroes on the left and right of the decimal point. That is, 4.2 and 4200000 and .00000042 all require 4 bytes to store. (This is not the case in packed decimal, for example, because significant zero is treated like any other digit.) The Number Class not only avoids needless padding with leading and trailing zeroes (in numbers like 00123 and .345000), but it also has a unique ability to compress significant zeroes (as in 10000 and .000006). Thus, Number Class shares the advantage of floating point's compact representation of very large and very small numbers, but without the drawback of floating point's limited accuracy.

Database Connectivity

If you build database applications, the Base/1 Number Class provides additional benefits whether you use a standard relational database management system (RDBMS), an object-oriented database, ISAM, or flat files to store your data. Besides its general compactness and speed, the Number Class addresses the following issues of importance to designers of information storage and retrieval systems:

Modern databases can store numbers that are longer than C++ can manipulate:

For example, Oracle can support a 38-digit numeric field, but there is no C++ data type that can exactly represent or perform arithmetic against such numbers. The Number Class can handle numbers of far greater length and precision than those supported by any major DBMS.

C++ numeric data types don't support the concept of a null number, distinct from zero:

In database applications, it can be useful to have a way of leaving a numeric field blank, meaning that data is not available, but the value is not necessarily zero. The ANSI SQL database standard has long provided for NULL values, but this notion is foreign to the C++ programming language. However, the Number Class does support null numbers, as distinct from numbers with a value of zero, so it handles this widely accepted database feature.

Conventional number representations are not amenable to efficient indexing:

Indexing is essential to good performance in large database applications, because it greatly speeds up searching and obtaining filtered, sorted query results. Unfortunately most numeric representations were designed for arithmetic, not for indexing. Consequently, extra logic is needed to transform numbers into keys in a format that collates properly. None of this logic is handled automatically by C++, so it usually falls upon the DBMS or the application programmer to perform the additional processing.

The Number Class is markedly different from conventional fixed or floating point binary numbers, which are inherently limited in their ability to represent exact decimal fractions. While resembling the "packed decimal" of mainframe computers, the Number Class uses a radically different encoding scheme to support much more efficient arithmetic and comparison algorithms. A unique benefit of Base One's encoding is that it makes numbers directly collatable as variable-length byte strings, regardless of the mix of numeric sign, scale, or precision. This makes the Number Class exceptionally well-suited to searching, sorting, and indexing applications.

The following table illustrates how Number Class values collate directly into proper numerical sequence, as conventional C signed character arrays. Unlike other numeric representations, the Number Class handles variable sign, precision, and scale, as well as a "Null" value with no need for conversion into a different representation for efficient sorting and indexing. Notice how the 10 sample numbers in the table collate correctly, from smallest to largest (null, -123.456, -12.3, -0.439, -0.0123, 0, 0.0123, 0.439, 1.234, and finally 123.456).

Collatability of Internal Representation

Number	Internal Representation as a Null-Terminated String (Byte Array Values)					
	1	2	3	4	5	6
(Null)	0					
-123.456	'n'	-12	-22	-44	-66	0
-12.3	'n'	-11	-22	-40	0	
-0.439	'n'	11	-53	-100	0	
-0.0123	'n'	12	-22	-40	0	
0	'o'	0				
0.0123	'p'	-12	22	40	0	
0.439	'p'	-11	53	100	0	
1.234	'p'	10	22	44	0	
123.456	'p'	12	22	44	66	0

The Number Class has a special internal string format that orders numbers properly under the rules of standard byte string comparison. In other words, first come blanks, then negative numbers, then zero, and finally progressively increasing positive numbers. No transformation is necessary for Number Class fields, because they are already in a representation that collates conveniently, making indexing extremely easy and efficient. Furthermore, the Number Class

allows compound keys to be formed simply by concatenating multiple fields, even if they are of mixed character and numeric types. No other numeric representation has these characteristics.

For users of BFC (Base/1 Foundation Component Library), a further advantage of the Number Class is its total integration with the Database Library's data dictionary and database retrieval and modification operations. Numbers can be stored into and retrieved from database records directly using Number Class variables.

In addition, Number Class instances can be initialized to the attributes of a database field defined in the data dictionary, including length (precision), scale, and a formatting mask to be used for display and/or data entry. For example, if you know that a calculated value will end up being stored in a particular field, initializing the number with that field's name will automatically insure that its length and scale will be the basis for proper rounding behavior and for generating overflow error notification when appropriate limits have been exceeded. (These are smart numbers!)

Comprehensive Error Handling

Even if your application doesn't require the highest level of precision, Base One's Number Class makes it easier to write reliable applications by adding convenient error-trapping features. This covers errors like Overflow, Divide by zero, Invalid sign, etc., which are not handled automatically by C++.

C++ provides only minimal facilities for detecting and handling errors that can occur in numeric arithmetic and conversion operations. This makes it necessary for programmers to take extra steps to be sure that certain errors, such as division by zero and overflow, cannot occur. Otherwise, a program may crash or simply yield incorrect results when one of these exceptional conditions arises.

All too often obscure bugs result from failing to add adequate error checking logic. The Number Class provides additional capabilities for detecting and handling errors, so these details don't burden the programmer but they also don't get overlooked:

The Number Class supports dynamic conversion between datatypes without requiring explicit casting. An error is thrown if an illegal conversion is attempted, for example a very large long value being assigned to a short int or a negative number to an unsigned. Exceptions are thrown whenever a significant truncation would take place or the number cannot sustain the required number of digits to the right of the decimal point without corrupting the number (e.g. ERR_OverflowNum, ERR_DivideByZero).

Convenient Overflow/Underflow Behavior:

If you add two conventional C++ numbers and the correct result requires more digits than the variable can hold, C++ doesn't detect the error, as you might have expected if you were accustomed to programming on old mainframe computers. Instead, C++ simply wraps the value, effectively truncating the most significant digits and producing totally incorrect results without any notification. Base One's Number Class gives you an easy way to trap and handle these errors, without having to write extra code to explicitly check for overflow on each arithmetic operation.

The behavior of Number Class arithmetic is analogous to arithmetic with native C++ datatypes with one useful difference. The Number Class throws an exception when destructive overflow errors occur.

For example in C++ if two short variables having +30,000 each are added, the result is not 60,000 and NO overflow error is reported. Similarly, if two short variables have -30,000 each and are added, the result would be incorrect and NO destructive "underflow" error would be reported. On the other hand, those same operations done with Number Class objects would result in exceptions being thrown that indicate the overflow error.

The Number Class throws these exceptions only if the number of digits on the left hand side of the decimal point of the result is too large to be held. If the number of digits on the right hand side of the decimal point is too large then it gets rounded according to the required scale.

When the result of arithmetic gets assigned to a Number Class object, the value gets converted to the precision and scale of the target Number Class. If the assignment would create a garbage result, an overflow exception is always thrown.

Numbers that are too small (insignificant) never cause exceptions to be thrown. Doing otherwise would be a pain. For example, addition of a number that is too small to have an effect clearly needs to be allowed (e.g. you have a scale of 2 and add .001). Assigning a number which is the result of some calculation that yields a very small number (one too small for the scale of the target Number Class variable) behaves as if the variable has been set to 0.

Division by zero:

C++ also doesn't detect invalid attempts to divide a number by zero (unless you consider immediately crashing to be a form of error "detection"). This too is a type of error that the Number Class makes easy to trap and handle cleanly, again without a lot of extra code. The Number Class also detects attempts to raise 0 to a negative power.

Conversion errors:

Another important category of errors is those that can occur when converting back and forth between different types of numeric and character representations. The limited error-handling features of C++'s conversion functions are greatly improved upon in the Base/1 Number Class.

For example, Number Class conversions discriminate among many specific error types, such as invalid sign, missing digit, negative number converted to an unsigned data type, particular flavors of overflow, etc. You also have the choice of detecting errors either based on a return code or by raising an exception for which you can supply your own handler.

Bottom line, you are guaranteed to be warned if the number is too large to be stored or if a desired degree of accuracy (precision) can't be maintained. Using the Number Class prevents the silent destructiveness of overflow that comes with all the C++ numeric data types.

But the main point is that Number Class numbers can be large - with many digits to both the left and the right of the decimal point. Having automatic support for larger ranges of numbers often means you avoid the silent damage caused by arithmetic truncation without any special effort dedicated to overflow error detection and handling.

Operating System Independent

To use the Number Class all you need is an ANSI-compatible C++ compiler. While the Number Class is a component of BFC (Base/1 Foundation Component Library), the class's modular design allows it to be distributed as a separate, standalone product. The Number Class does not depend on BFC. If you have BFC, the Number Class is included.

The Base/1 Number Class is compatible with both the Microsoft Foundation Classes (MFC) and the Standard Template Library (STL). You can use the Number Class whether you write Windows/MFC programs with Microsoft's Visual C++ or you program for an STL environment like Unix, using ANSI/C++. The sample code that ships with the product includes MFC and STL based projects, which both execute the same examples of Number Class usage.

The Base/1 Number Class is independent of all platforms and operating systems because it is written in ANSI standard C++ and can use the Standard Template Library (STL). The Number Class can be used in various environments like DOS, Windows, UNIX, OS/390 etc., all of which have a standard ANSI/C++ compiler.

(We've been careful to use only the most basic C++ features and the standard, public STL interface. In addition, we have upgraded the source code whenever developers have reported minor compilation errors when working with non Microsoft compilers. With the Number Class source code, we supply a small string class, clsStrPlus, as a layer between the Number Class and raw MFC and STL strings, and this string class can be easily adapted to handle any peculiarities of the particular STL implementation being used.)

Easy To Use

Thanks to operator overloading, using the Base/1 Number Class is virtually identical to using built-in C++ number types. The usual arithmetic operators, assignment statements, parameter

passing, etc. work with Number Class variables just as they work with standard C++ numbers. And, because of Reference Counting, the overhead of using the Number Class is comparable to that of standard C++ data types, so it can be used freely without having to be conscious that it is a user-defined C++ class. But the Number Class has significant advantages over native C++ number types due to these fundamental differences:

Number Class as a String:

The Number Class (clsNumDbFld) has the convenience of allowing a number to be handled as a string, as well as a number. Using a string provides a way to specify constants that are outside the allowable range of C++ numeric constants.

Here are samples of valid assignment statements for the class -

```
clsNumDbFld numTest;
numTest = 123;
numTest = 1.23;
numTest = -123.456;
numTest = "123";
numTest = "12345678901234567890134567890";
numTest = 0;
numTest = "0";
numTest = ""; // null
```

Number class objects can be assigned numeric string values in standard null-terminated C format and can return numeric string values. For example, a number stored in a Number Class object can be returned as a string by doing a cast operation using MFC CString.

Here's an example of initializing a Number Class with a string and then verifying that its string value is returned as entered:

```
clsNumDbFld numTmp( "1.12345" );
CString strTmp = "1.12345";
ASSERT( strTmp != (CString) numTmp ); // Never asserts. Always equal
```

This code works in the STL implementation because CString is always treated as string, the result of a typedef in the supplied String Plus class header.

Instead of CString, you can use the String Plus class (clsStrPlus), which is designed to work with MFC's CString or STL's string. The String Plus class adds convenient conversion and assignment routines, trimming, padding, concatenation etc. Here's an example, that handles insignificant left and right 0's correctly.

```
clsNumDbFld numTmp( "1.123456789" );
clsStrPlus strTmp = "01.12345679000"; // insignifcant 0's stripped
ASSERT( strTmp != (clsStrPlus) numTmp ); // Never asserts. Always equal
```

The Number Class makes null numbers equivalent to null strings, i.e. the first byte is zero ('\0'). IsNull(), which checks if a string is empty or BLANK, is an example of a useful function inherited from clsStrPlus. Arithmetic operations on null numbers have the same intuitive behavior as the handling of blanks by spreadsheet programs. That is, null is always treated as ZERO in arithmetic (and NOT as "unknown").

Rounding Behavior of the Number Class:

The Number Class handles rounding like a well behaved, floating point number. The last significant digit is rounded up if the next digit is greater than or equal to 5. For example, 1/3 = 0.33333333 but 2/3 = 0.66666667.

When a Number Class has a larger scale (more digits to the right of the decimal point), rounding (NOT trunction) takes place. The RoundNum() function rounds a Number Class's currently held value to a specified scale.

Different forms of rounding rules are not needed because of the wide range of numbers supported. With 100 digits of precision available, it's easiest just to adjust the scale (from the default of 8 digits to the right of the decimal point) to one that's large enough so that rounding inaccuracies can be ignored.

Overloaded C++ Operators:

The Number Class can be used just like a native C++ datatype, with the full spectrum of operators and in all the kinds of numeric expressions supported by C++. The arithmetic operators (+ - * / %) and the comparison operators (== != > >= < <=) are implemented as global overloaded operators in the header file for Number Class inline functions.

For example, the Compare Value function (CompVal) need not be called explicitly for doing comparison. An overloaded comparison operators can be used, which, in turn, causes an inline function call to CompVal().

These operators have to be global. If they were implemented as member functions and not global functions, then the operand on the left hand side would always have to be Number Class object. That would mean that "num1 > 2" would be legal but "2 < num1" would not be. As global functions, both situations are covered by having two functions, the first function take "num1, short" for example and the second function takes "short, num1".

Avoiding Garbage from Floating Point Constants:

Be aware that when you do assignments to a Number Class objects from floating point constants that you may notice some very small garbage far to the right of the decimal point. In fact, whether that constant is written in scientific notation or standard decimal notation will affect the results. (The different tiny garbage values are due to the low-level C Runtime Library or possibly the hardware floating point unit itself.)

For example -

```
clsNumDbFld num1( 100, 50 ), num2( 100, 50 ), num3( 100, 50 );

num1 = 9.234e+2;
num2 = 9.234e-2;
num3 = num1 / num2;
num3 equals 9999.999999999999

num1 = 923.4;
num2 = .009234;
num3 = num1 / num2;
num3 equals 10000.0000000000000649
```

In both cases, of course, the value of num3 should be exactly 10,000. To avoid the problem with floating point constants, you can always express the values as "923.4" and ".09234" (string literals) and guarantee accuracy. By expressing decimal numbers without tiny bits of garbage being carried, the Number Class makes extreme precision possible.

Speedy Performance

Besides simply saving space, the compactness of Base One's Number Class representation provides a performance boost by shortening in-memory data transfers and reducing the amount of disk I/O. But there are additional reasons for significant speedups owing to the unique design of the Base/1 Number Class:

All arithmetic calculations are performed taking two digits at a time:

Base One's algorithms for operations like addition, subtraction, multiplication, division, remainder, quotient, modulus, etc. have been designed to process digits in a pairwise fashion. On the other hand, Binary Coded Decimal (BCD), which also packs two digits into each byte, requires twice as much work to extract and process each digit individually.

Extremely fast numeric comparison logic because no normalization is required:

A remarkable feature of the Number Class is that it uses highly efficient byte string operations (exactly like strcmp) to directly compare any pair of numbers. Even if the numbers differ in sign, length, or scale, there is no need for the usual "normalization" step to precede comparison or arithmetic operations. Other numeric encodings (such as BCD) require a great deal of additional processing, digit by digit, to put both numbers into a common intermediate representation before their values can be operated upon. This distinction makes Base One's Number Class much better suited than other technologies for building high-speed searching and sorting applications.

Reduced memory management overhead:

The benefits of "Reference Counting" (discussed above), not only mean that less memory is consumed, but also less processing is required for dynamically allocating and freeing memory when Number Class values are copied between variables. What's more, this memory management logic is implicitly handled by the robust, efficient, underlying STL/MFC string classes.

Performance Limitations:

The Number Class is slower than double arithmetic because operations are done exclusively through software versus the hardware implementation used for double arithmetic. (Floating point simulation through software is a thing of the past.) Fortunately, tight loops, such as used for simple-minded benchmarks, most often do NOT accurately represent the actual demands on the Number Class.

The Number Class is particularly well suited to business systems, which are generally I/O bound, and good results are usually obtained with a blanket substitution of the Number Class for ALL floating point and integer variables. On the other hand, the Number Class is NOT optimized for the kind of number crunching done, for example, in large scale matrix operations.

Benchmarks are difficult to make realistic or relevant and difficult to interpret, so we never publish benchmark results. Test results are easily skewed by the way the benchmark is constructed. For example, the order of the tests can slant results because of low-level caching (such as instruction caching, file caching, and virtual memory paging).

In some cases, to achieve acceptable performance, you may find it necessary to play games to get the benefit of the Number Class accuracy and the speed of hardware-based floating point arithmetic. That is, you can check programmatically whether you need to use a double and double arithmetic versus the possibly smaller number of instances where you need to use the Number Class and its arithmetic. (Remember to do your performance testing in RELEASE mode and NOT in Debug mode.)

Logarithmic, Trigonometric, Exponentiation Functions:

The current version of Number Class handles the basic arithmetic operations but does not handle logs, exponentiation (except for positive and negative integral exponents) or trigonometric functions. This is usually NOT a problem because you can intermix standard C++ function calls with Number Class variables. Casting (to double, for example) takes place automatically. (We're still trying to uncover an efficient algorithm for determining values to a specified precision, such as base e raised to any decimal power.)

Algorithms Used in Number Class Arithmetic

The Number Class uses an innovative variable-length encoding of numbers that bears some resemblance to Binary Coded Decimal (BCD), but is considerably more efficient in several respects. One of the keys to Base One's unique design is its use of fast, robust STL/MFC string classes to maintain the internal representation of numbers. This design makes possible a number of features not found in other numeric encoding technologies:

- all arithmetic calculations are performed taking two digits at a time

- fast byte-by-byte comparison logic requires no normalization to handle variable sign, precision, or scale

- no explicit length field is required, reducing both size and processing complexity

- significant zeroes are compressed, further reducing storage requirements

- compact, efficient memory allocation logic is inherited from the standard STL/MFC string classes

- direct collatability of signed numbers of varying length and scale by efficient byte string comparison

Exact arithmetic involving very large (or small) numbers is automatically supported and arithmetic can still be done using the convenient, familiar arithmetic operators. The Number class also directly supports the database concept of a null number (all blank), as distinct from a number with a value of zero. (Nulls are not permitted in any C++ numeric data types.)

The following diagrams illustrate the algorithms for addition, subtraction, multiplication and division. Unbiased byte values are shown for the sake of clarity, however the actual implementation operates directly upon the biased values of the Number Class internal representation.

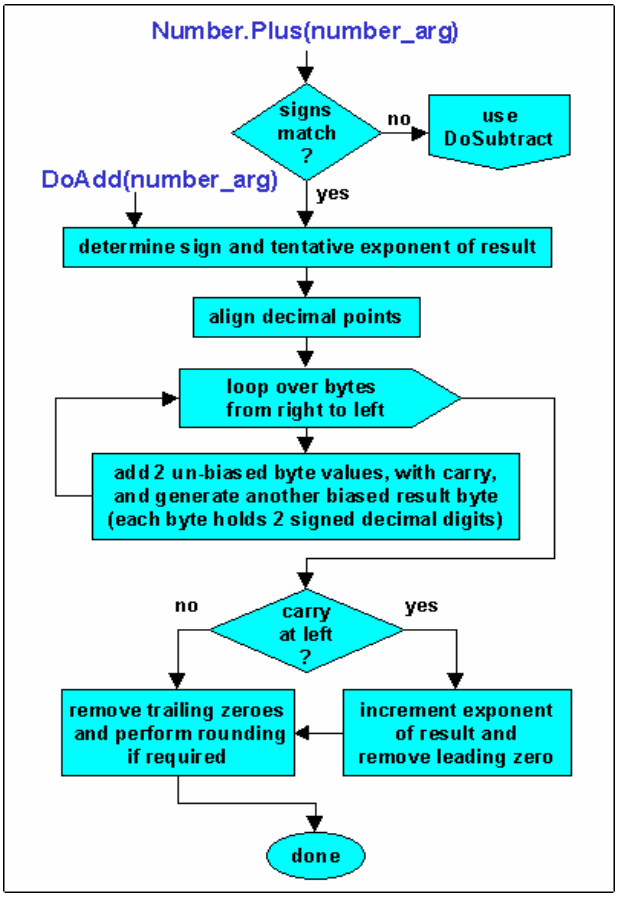
Number Class Addition Algorithm

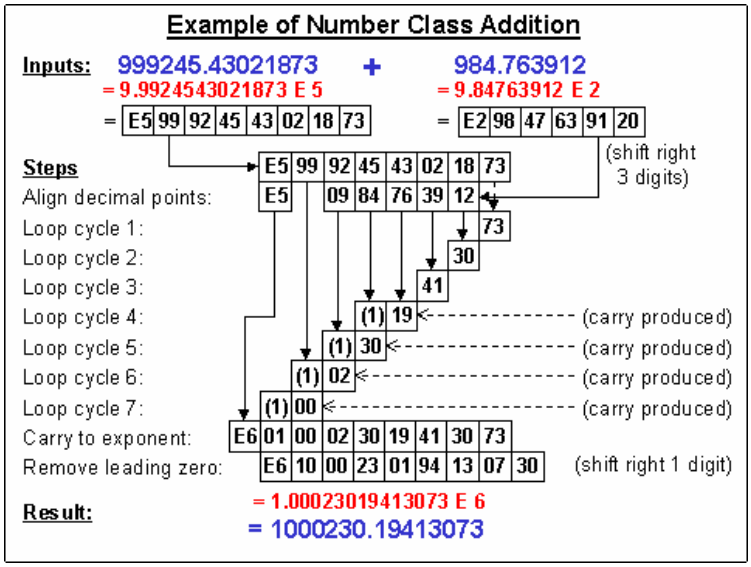
In brief, addition works as follows -

First, the decimal points are aligned by increasing the lesser number's exponent and logically shifting its mantissa to the right by the corresponding number of digits. For the sake of efficiency, a byte offset is maintained to avoid physically shifting by whole bytes. If the difference in exponents is odd, an additional single-digit physical shift is performed on the lesser number.

As with all of the Number Class algorithms, the actual computation is performed in units of bytes, i.e. processing two decimal digits at a time from each of the operands. The addition is performed starting from the rightmost end, and the bytes are added with carry, if any. If there is a

carry at the leftmost end, then the result exponent is incremented by one, and a single-digit shift is applied to remove the leading zero. Finally, any trailing zero bytes are removed, and rounding is done if the result's scale exceeds the scale of the number being operated upon.



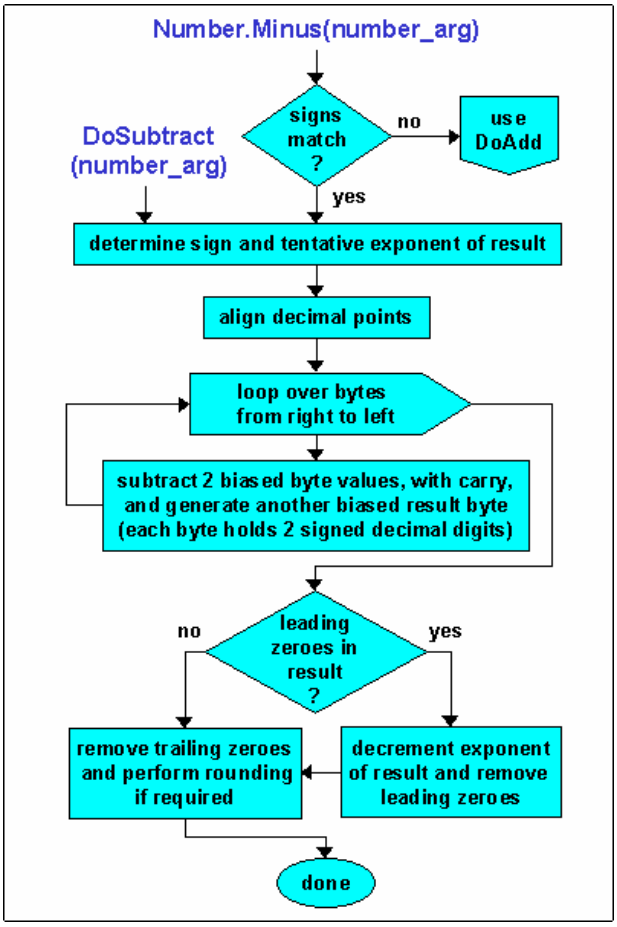


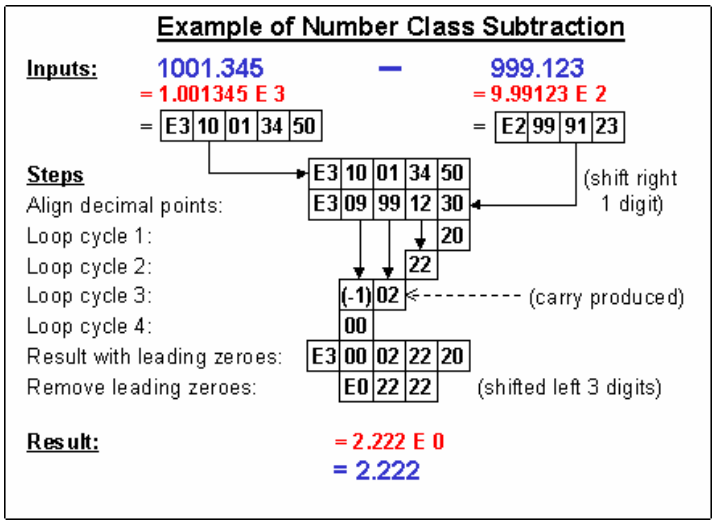
Number Class Subtraction Algorithm

In brief, subtraction works as follows -

First, the decimal points are aligned by increasing the lesser (in absolute value) number's exponent and logically shifting its mantissa to the right by the corresponding number of digits. For the sake of efficiency, a byte offset is maintained to avoid physically shifting by whole bytes. If the difference in exponents is odd, an additional single- digit physical shift is performed on the lesser number.

As with all the Number Class algorithms, the actual computation is performed in units of bytes, i.e. processing two decimal digits at a time from each of the operands. The subtraction is performed starting from the rightmost end, and the bytes are subtracted with carry, if any. Since the lesser number is always subtracted from the greater, there is never a carry beyond the leftmost digit. Any leading zero digits are removed by logically shifting the mantissa leftward and correspondingly decrementing the result's exponent. Finally, any trailing zero bytes are removed, and rounding is done if the result's scale exceeds the scale of the number being operated upon.



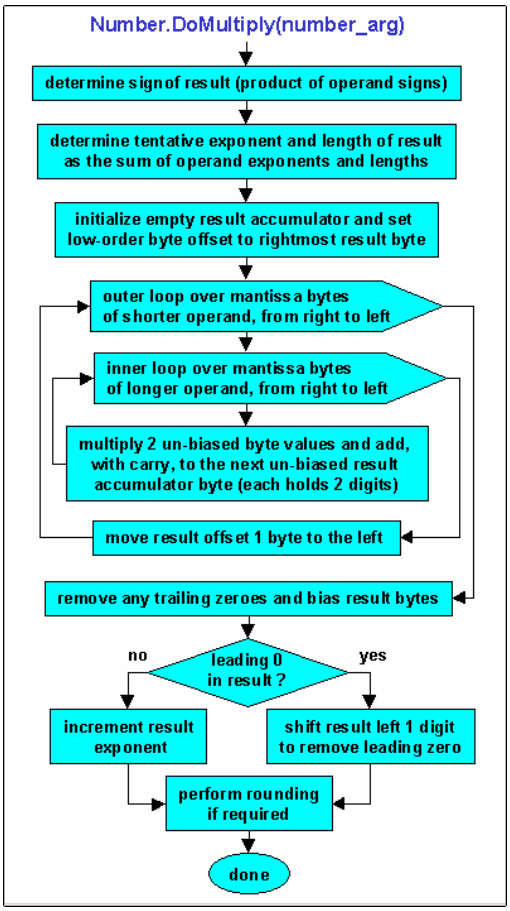


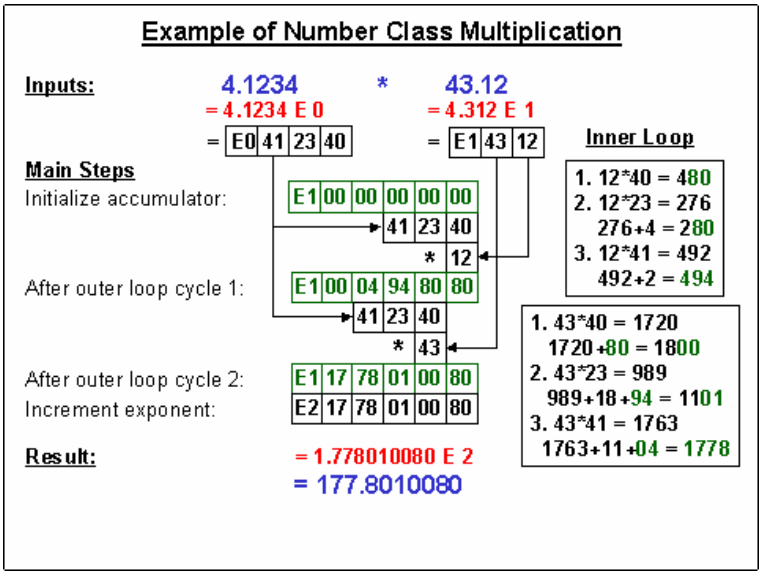
Number Class Multiplication Algorithm

In brief, multiplication works as follows -

First, the result's sign, maximum length, and tentative exponent are determined, and a buffer of sufficient size is allocated and initialized with zeroes. This buffer is used to accumulate the results of each loop cycle with the un-biased byte values and adds this result to the next result accumulator byte, again processing from right to left. With each successive outer loop cycle the offset of the lowest order (rightmost) result byte computed moves leftward by one byte. (The accumulator holds un-biased byte values while processing is being performed.)

Trailing zeroes are always removed, except at most a final zero digit if needed to fill out the last full byte. If the final result accumulator contains a leading zero, this is removed by shifting the mantissa left by a single digit. Otherwise, the tentative exponent is incremented, and the exponent is incremented, and the exponent's final sign and bias are applied. Rounding is done if the result's scale exceeds the scale of the number being operated upon.





Number Class Division Algorithm

In brief, division works as follows -

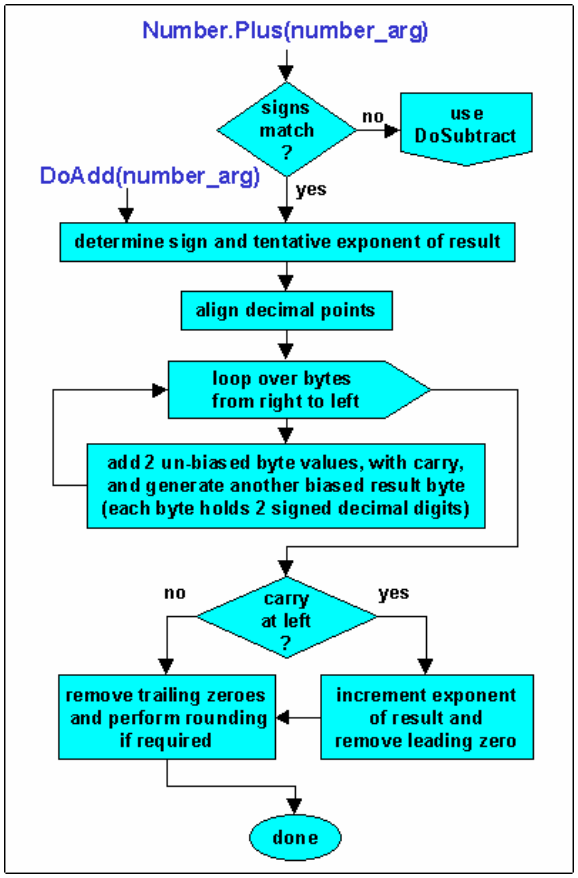
DoDivide() supports either division or a quotient and/or remainder (modulo) function, depending on whether non-null arguments are supplied. For the quotient and remainder cases, the logic will generarly terminate sooner than full division. (The diagram only shows the details of the division case.)

First, the signs of the result, quotient, and remainder are determined. The result's (and quotient's) sign is positive if both the operands have the same sign, otherwise it is negative, and the sign of the remainder is always that of the dividend, i.e. the number being operated upon. The result's exponent is calculated as the difference of the exponents of the dividend and the divisor argument. A maximum number of result mantissa digits is calculated by adding the result's exponent, plus an allowance for the Number Class variable's declared scale factor (i.e. digits to be retained to the right of the decimal point), plus an extra digit to support proper rounding to the desired scale.

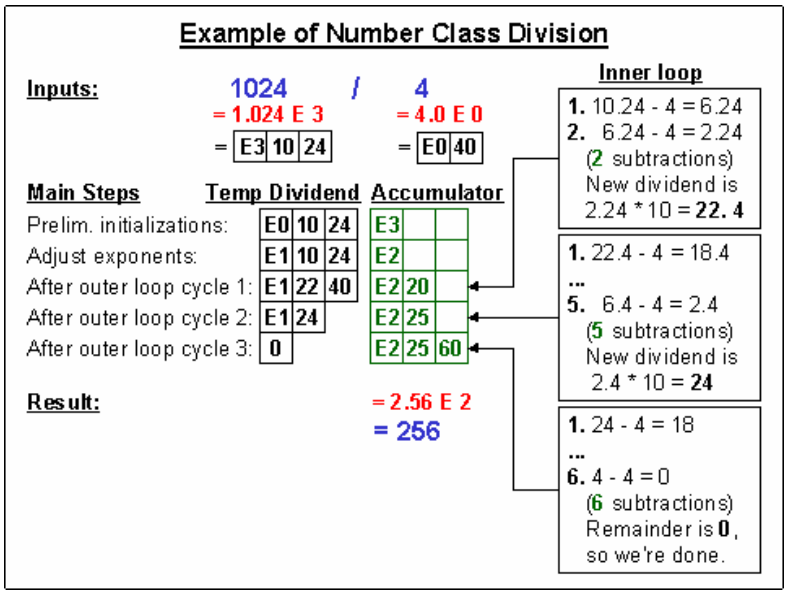
A result accumulator of sufficient length is allocated and the dividend is copied into this temporary Number Class variable, which will be used as the remaining dividend in the following logic. (For the special case of a quotient/remainder operation, however, the calculation immediately finishes at this point if the divisor is larger than the dividend.) The exponent of the temporary dividend is then adjusted to be the same as that of the divisor. If this leaves the dividend smaller than the divisor, the dividend's exponent is incremented and the result's exponent is decremented, as the final initialization step. (The accumulator is generated with biased mantissa bytes as each result digit is obtained.)

The main loop generates one mantissa digit on each cycle, filling the result accumulator from left to right. Each result digit is computed by an inner loop that subtracts the divisor from the dividend, until the remaining dividend is less than (in absolute value) the divisor. Note that these comparison and subtraction operations are performed via Number Class functions, CompAbsVal() and DoSubtract(), since the divisor and dividend are both Number Class objects. The number of subtractions performed by the inner loop yields the next digit of the result. If the remainder is zero, the outer loop terminates; otherwise the remaining dividend is multiplied by 10 and processing for the next result digit continues, up to the required scale and precision. (If obtaining only quotient/remainder, exit when we have calculated all the result digits to the left of the decimal point, i.e. the quotient.)

Trailing zeroes are always removed, except at most a last zero digit if needed to fill out the last full byte. A final rounding step is done only if the result's scale exceeds the scale of the number being operated upon.



Note that the example (1024 / 4) illustrates an instance where the initial temporary dividend (1.024), after forcing its exponent to that of the divisor (0), is smaller than the divisor (4.0), so the dividend's exponent is incremented and the result's exponent is decremented. Thus, the outer loop logic begins it processing against a dividend of 10.24.



Appendix

Microsoft Modulo Bugs

Over the years of testing of the Number Class, we've discovered nasty bugs in Microsoft's fmod() function, which you can easily confirm for yourself. For example, using the Windows 95 Calculator accessory (evidently built on the same fmod function) try this simple calculation:

5.5 mod 1.1

Instead of giving the correct result, which should be 0, it shows the result as 1.1. Similar errors are not hard to find, e.g. try 5.1 mod 1.7, 49 mod 9.8, 21.9 mod 7.3, and 36 mod 7.2. Interestingly, this problem occurs only with certain choices of values and the pattern of failures is not trivially discerned, but there are obviously many more cases. In fact we found it increasingly rare not to encounter such errors with more decimal places. We were unable to find any reference to this disturbing bug in the Microsoft Knowledge Base.

Considering how much flack Intel took over an even more obscure arithmetic bug in its Pentium processor, one can only wonder how Microsoft ever got away with this one. Fortunately, the problem seems to have been quietly fixed in Windows 98, but that's not much consolation to those of us who expect our programs to work properly under Windows 95 and early versions of Windows NT 4.0. In any case, rest assured that the Base/1 Number Class computes the modulo function correctly under any version of Windows.

The modulo bug described above seems to have been fixed in the latest operating systems and service packs from Microsoft. However, here's a new one we discovered in recent testing of the Number Class:

```
d1 = 8853.41959899;
d2 = 2951.13986633;
```

fmod(d1, d2) returns the same value as that of d2 instead of returning zero. In this example, d1 is a perfect multiple of d2 (d1 = d2 * 3). We got this error on Windows NT Service Pack 5 and 6a and Windows 2000 server. For some versions, the bug did NOT appear in the calculator program, but using fmod() ALWAYS produces this bug. The Number Class properly returns the correct remainder of zero.