;login:

THE MAGAZINE OF USENIX & SAGE
April 2002 • Volume 27 • Number 2

inside:

USENIX & SAGE
The Advanced Computing Systems Association &
The System Administrators Guild

math library functions in C9X

We've been discussing some of the new features in C9X, the standards update to C. In this column we'll look at new math functions and, in particular, the philosophy behind the inclusion of the functions in the library.

by Glen McCluskey

Glen McCluskey is a consultant with 20 years of experience and has focused on programming languages since 1988. He specializes in Java and C++ per-for-mance, testing, and technical documentation areas.

glenm@glenmccl.com

Why So Many Functions?

There are lots of new math functions in C9X. One reason for the proliferation is that many functions have been generalized to work with float and long double types, in addition to the double type already supported. Here's an example:

```
#include <math.h>
#include <stdio.h>

const long double CONV_DEG_TO_RAD = 360.0L / (2.0L * 3.14159265L);

int main()
{
    long double d = sinl(90.0L / CONV_DEG_TO_RAD);

    printf("%Lg\n", d);
}
```

C has always had a sin() function, but now it also has sinf() and sinl() functions, which operate on float and long double types.

But this is only part of the picture. C9X also includes a cbrt() function, which calculates cube roots. Why is such a function needed? Why couldn't you simply say:

```
double res = pow(val, 1.0/3.0);
```

that is, compute the cube root by taking the 1/3 power of a value?

The answer is that you can, but cbrt() offers some advantages. One is that it will work on negative numbers: for example, -27.0, with a cube root of -3.0. Another advantage is that there may be a more efficient means of calculating cube root than pow(), or an algorithm that has better error properties. For example, when we run this program:

```
#include <math.h>
#include <stdio.h>

double get_cube_root1(double d)
{
    return pow(d, 1.0/3.0);
}

double get_cube_root2(double d)
{
    return cbrt(d);
}

int main()
{
    double d = 259.0;
    double res, diff;

    // get cube root using pow()

    res = get_cube_root1(d);
    diff = res * res * res - d;
    printf("diff using pow() = %g\n", diff);
```

```
        // get cube root using cbrt()

        res = get_cube_root2(d);
        diff = res * res * res - d;
        printf("diff using cbrt() = %g\n", diff);
    }
```

we get the following result:

```
    diff using pow() = -1.09746e-13

    diff using cbrt() = -1.47105e-15
```

indicating that cbrt() has less error than pow(). Note also that the 1.0/3.0 we passed to pow() does not necessarily exactly represent 1/3, given that many fractions are not exactly representable in IEEE floating-point format.

Here's another example of why you might want to use a library function to perform an "obvious" calculation. Suppose that you're computing the square root of a sum of squares, like this:

```
    sqrt(X*X + Y*Y)
```

C9X provides a function hypot() to perform this calculation. Why? Here's one example where it matters:

```
    #include <float.h>
    #include <math.h>
    #include <stdio.h>

    int main()
    {
        // set up two constants equal to sqrt(DBL_MAX)

        double x = sqrt(DBL_MAX);
        double y = x;

        // compute square root of sum of squares

        double z = sqrt(x * x + y * y);
        printf("%g\n", z);

        // same computation using hypot()

        z = hypot(x, y);
        printf("%g\n", z);
    }
```

The result of running this program is:

```
    inf

    1.89615e+154
```

x and y have values equal to sqrt(DBL_MAX). When these values are squared and then added, the result is an overflow – that is, infinity.

But hypot() works in this case. The C9X specification directly addresses the intermediate overflow issue:

> The hypot functions compute the square root of the sum of the squares of x and y, without undue overflow or underflow.

One final example. The library contains an fma(x,y,z) function, that computes x*y+z. Why would such a simple calculation require a library function? Once again, we look to the C9X specification for a clue:

> The fma functions compute (x*y)+z, rounded as one ternary operation; they compute the value (as if) to infinite precision and round once to the result format, according to the rounding mode characterized by the value of FLT_ROUNDS.

The C9X rationale document further says that there is often hardware support for the fma() function, that is, an instruction that does a floating multiply and add.

These examples illustrate the point that many of the new functions are specifically designed to meet the demands of numerical programming. Whether all these functions fit within the "Spirit of C" depends on your philosophy, but they certainly move the language toward the goal of supporting numerical programming applications.

Classifying Numbers

We'll now look at a few additional examples of new functions. One new area is number classification, where you can determine whether a given number is normal, subnormal (numbers with a minimum exponent and leading zero bit for the fraction part), infinite, NaN, or zero. Here's an example:

```
#include <float.h>
#include <math.h>
#include <stdio.h>

void classify(double d)
{
    // print the number

    printf("%g\t", d);

    // print the classification of the number

    switch (fpclassify(d)) {
        case FP_ZERO:
            printf("zero\n");
            break;
        case FP_NORMAL:
            printf("normal\n");
            break;
        case FP_SUBNORMAL:
            printf('subnormal\n");
            break;
        case FP_INFINITE:
            printf("infinite\n");
            break;
        case FP_NAN:
            printf("NaN\n");
            break;
    }
}

int main()
{
    double d1 = 12.34;
    double d2 = 1.0 / 0.0;
```

```
        double d3 = log(-1.0);

        // regular number

        classify(d1);

        // infinity

        classify(d2);

        // NaN

        classify(d3);
    }
```

One of the fundamental issues with numerical programming is how to handle cases where calculations overflow, or situations where an "impossible" operation is attempted, such as taking the log of a negative number. As part of handling these cases, values that are not normal numbers need to be represented: for example, positive infinity or NaN.

Gamma Functions

tgamma() and lgamma() represent another example of new functions. These compute the gamma function and the log of the gamma function. This function is related to factorial, in that:

```
n! = gamma(n + 1)
```

but the gamma function is a more general mathematical concept.

Here's an example of using these functions to compute the number of permutations of N objects taken R at a time:

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
    if (argc != 3) {
        fprintf(stderr, "Usage: %s N R\n", argv[0]);
        return 1;
    }

    // get N and R

    int n = atoi(argv[1]);
    int r = atoi(argv[2]);

    // compute permutations using tgamma

    double num1 = tgamma(n + 1);
    double denom1 = tgamma((n - r) + 1);
    double res1 = num1 / denom1;

    // compute permutations using lgamma

    double num2 = lgamma(n + 1);
    double denom2 = lgamma((n - r) + 1);
    double res2 = exp(num2 - denom2);
    printf("result using tgamma = %g\n", res1);
    printf("result using lgamma = %g\n", res2);
}
```

For command line arguments of 10 and 5, the result is 30,240, that is, if you have 10 objects taken 5 at a time, then there are 30,240 possible permutations. The formula for permutations is:

```
N! / (N-R)!
```

The tgamma() function value grows very rapidly, and sometimes you might wish to use lgamma() instead. We've illustrated how to perform the same calculation using logs and exp().

Rounding the Results of Calculations

A final example illustrates the many functions that you can use for rounding to the nearest integer. "Nearest" has multiple definitions, as this demo makes clear:

```
#include <fenv.h>
#include <math.h>
#include <stdio.h>

int main()
{
    // smallest integer not less than the argument
    printf("%g\n", ceil(2.4));

    // largest integer not greater than the argument
    printf("%g\n", floor(2.4));

    // set rounding direction to upwards
    fesetround(FE_UPWARD);

    // round to nearby (upwards) int, and return a double
    printf("%g\n", nearbyint(2.4));

    // round to nearby (upwards) int, and return an int
    printf("%ld\n", lrint(2.4));

    // always round halfway cases upward
    printf("%g\n", round(2.5));

    // round to nearest integer not larger than the argument
    printf("%g\n", trunc(2.9));
}
```

The output of the program is:

```
3
2
3
3
3
2
```

We've tried to present some of the philosophy around the many new C9X math functions. These features represent a major step forward for C as a numerical and scientific programming language.