Fixed-Point Arithmetic Types for C++

by Kurt Guntheroth

For version 2.1 of the fixed-point library

Computers cannot exactly represent the value of fractions such as 8/9, which aren't evenly divisible by 2 [1]. In fact, the value of the integer expression `8/9` is zero, even though the mathematical quantity is much closer to 1. For computations involving fractions, most developers use floating-point arithmetic, which approximates the value of 8/9 as a decimal fraction like 0.88889, a more sensible and accurate choice than the unintuitive result of the integer expression.

Floating-point numeric representation is internally complex. This makes floating-point math slower than integer math, even when floating-point instructions are built into the processor. Fortunately, there are other ways to represent fractional numbers.

A fixed-point number (for instance, the decimal fixed-point number 123.456) is written with a radix point and a fixed number of digits to the right, representing a fractional part. We say "radix point" instead of decimal point, because both decimal and binary numbers may usefully be represented in fixed-point form. In a decimal fixed-point number, the digits to the right of the radix point are decimal digits. In binary fixed-point, the fractional digits are binary bits. A decimal fixed-point number can be exactly represented as an integer by multiplying it by a scale factor that is a power of 10, effectively moving the decimal point to the right. A binary fixed-point number is represented in the same way, except the scale factor is a power of 2. Fixed-point numbers may thus be implemented using integer arithmetic operations that execute faster than corresponding floating-point operations.

The precision of a fixed-point number (the difference between one value and the next bigger representable value) is the same all the way through its range. By contrast, half of all representable floating point numbers lie between 0 and 1, and consecutive values grow further apart toward the large end of their range. Decimal fixed-point numbers can exactly represent decimal fractions like dollars and cents, whereas conventional floating-point numbers cannot. Some financial computations that produce inexact results using floating-point, produce exact answers using decimal floating point. The range of a fixed-point type is limited by the range of the integer which represents it internally, whereas floating point numbers can represent extremely large numbers [2]. For a large class of everyday problems that don't need the astronomical values representable by floating-point numbers, this tradeoff is quite reasonable.

I coded C++ template classes implementing three fixed-point types. They are intended as drop-in replacements for `int` or `double`. Their precision and range can be tuned to suit different applications. On my Pentium 4, compiled by Visual C++ 7.1 with optimization turned on, the fixed-point types presented here are over twice as fast as equivalent scalar floating-point code. On embedded processors and integer Digital Signal Processor chips with no floating-point hardware, I expect them to be many times faster than floating-point function libraries.

Arithmetic Helper Template Class

The fixed-point classes directly implement four arithmetic assignment operators (+=, -=, *=, /=), and two comparison operators (==, <). A trick from the Boost numerics library [3] uses a helper class to define five more arithmetical operators (+, -, *, /, unary -) and the remaining comparison operators (!=, >=, <=, >). The fixed-point classes derive from the helper template class, which takes the fixed-point class as a template argument. At first glance, this technique looks impossibly circular. It works because the compiler doesn't check template functions until they are instantiated, and both classes have complete definitions at this point. Listing 1 illustrates how the helper template implements `operator+()` and `operator>=()` in terms of the derived class's `operator+=()` and `operator<()` respectively.

[Listing 1: helper class example]

```
template <class D> class Helper
{
    friend D operator+(D const& lhs, D const& rhs)
    { D tmp = lhs; tmp += rhs; return tmp; }

    friend bool operator>=(D const& lhs, D const& rhs)
    { return !(lhs < rhs); }
};
class D : public Helper<D>
{
    int rep_;
public:
    D(int i=0) : rep_(i) {/*empty*/}

    D& operator+=(D const& rhs)
    { rep_ += rhs.rep_; return *this; }

    bool operator<(D const& rhs)
    { return rep_ < rhs.rep_; }
};
```

Chip makers have spent much effort and silicon (perhaps 20 per cent of the Pentium 4 die) to perform loating-point computations quickly. Beating the Pentium's floating-point performance in my fixed-point types was thus a non-trivial challenge. I implemented C++ template classes so the compiler would inline method-calls. With the code inlined, the compiler has many optimization opportunities. Mainstream compilers produce very impressive optimization results on this code.

I benchmarked the fixed-point implementations against floating-point code, and checked the compiler-generated assembly-language output to see if unexpected artifacts were confounding my measurements. The full code listing, available on-line, contains a tool for measuring the relative performance of the fixed-point classes versus floating-point on a sample problem involving multiplication and division.

Because different compilers implement different optimizations, users of these fixed-point classes may need to tweak them for best performance on other compilers or non-Intel processors. The Boost numerics library [3] contains an alternate implementation for compilers that don't perform the returned value optimization.

Rounding Integer Template Class

The rounding integer representation produces a more satisfying integer result by rounding the result to the nearest integer, rather than truncating it toward zero. Thus the value of the expression 8/9 in rounding integer representation is 1, not 0. The speed penalty (on my Pentium 4 using VC7) is only 15 per cent versus ordinary integer calculations involving division. Rounding integers have the same range as ordinary integers.

The rounding integer representation considers an integer as a binary fixed-point number with no bits to the right of the radix point. Integer addition, subtraction, and multiplication produce exact results. They are implemented using the arithmetic operations for the underlying integral type.

Integer division truncates toward zero. To round the result of integer division, the code takes advantage of the C++ modulus (`%`) operator. The value of `a%b` is the integer remainder of the expression `a/b`. If the remainder is greater than or equal to half the divisor, the quotient is rounded up by adding 1. Rather than divide the divisor by 2 (reintroducing the undesirable truncation) I double the remainder (using left shift for efficiency) and compare it to the divisor.

```
q = n / d;
r = n % d;
if ((r<<1) >= d) q += 1;
```

Multiplication and division are computationally expensive. However, the integer division instruction on most processors produces both a quotient and remainder. The Visual C++ optimizer efficiently reuses the already-computed remainder in the situation above instead of recomputing it. On more modest hardware, or less-optimizing compilers, the standard C runtime function `ldiv()` can be used to produce a structure containing both quotient and remainder. `ldiv()` is not an intrinsic function in VC7, so it has a high performance cost versus relying on the optimizer.

The signs of the operands present a further challenge in rounding. The signs of the quotient and remainder depend on the signs of the divisor and dividend. Handling all four cases of operand sign generates some code bloat, but not much speed penalty. I have verified this code on Intel-architecture processors. Other processors implement the sign of the remainder differently, so users of this template should verify its correctness with their processor. The full code, available for download, contains a test to ensure that the implementation works correctly on a given processor and compiler.

I investigated a branch-free rounding computation to improve performance, because branching causes pipeline stalls on some processors. On the Pentium 4 there was no observable improvement, though this may be due to optimization.

Listing 2 shows the division code with rounding.

[Listing 2: division with rounding]

Decimal Fixed-Point Template Class

In the decimal fixed-point template class, a template parameter `N` gives the number of decimal digits to the right of the decimal point. The decimal fixed-point type precisely represents all decimal fractions (all hundredths, thousandths, etc., depending on the template parameter). It is an excellent choice for computations involving currency because it exactly represents every penny. Whole integer values are represented internally by multiplying the integer by a scale factor that is the Nth power of 10; 10^2 or 100 for dollars-and-cents calculations. Thus if N equals 2, the value 1.23 has an exact representation as the integer 123.

Addition and subtraction are implemented as ordinary integer operations on the scaled values. Addition and subtraction always have exact results.

We multiply decimal fixed-point numbers using integer multiplication to obtain an intermediate product. The intermediate product has twice as many fractional digits as the operands, so if the operands must have the full 32 bits of significance, the intermediate result requires 64 bits. An implementation can use a `long long` variable for the intermediate product, or accept overflow over a wider range of operands. The intermediate product is divided by the scale factor to produce the proper number of fractional digits in the final result. This is the familiar shift-the-decimal-point operation from the elementary-school multiplication algorithm for decimal numbers. The scaled result is rounded if the remainder is greater than or equal to half the scale factor.

For division, we multiply the dividend by the scale factor before dividing. This leaves the quotient with the proper number of decimal digits of precision. As with multiplication, the implementation must choose between using a `long long` intermediate dividend, or accepting overflow on a greater range of dividends. The quotient is rounded up if the remainder is greater than or equal to half the scale factor.

Multiplication by an integer is always exact. Other multiplications and all divisions may involve rounding, so it is incorrect to assume that decimal fixed-point will make all your inexact computations go away. Still for financial computations, the result is always rounded to the nearest penny. This frequently produces the desired result.

Listing 3 shows multiplication and division code for decimal fixed-point numbers. I implemented scaling the intermediate product in terms of rounding division to avoid duplicating the tricky rounding code.

[Listing 3: fixed decimal multiply and divide]

```
fixed_decimal& operator*=(fixed_decimal const& that)
{//   multiplication with rounding
      this->rep_ = scaled_multiplication(this->rep_, that.rep_,
scale_);
      return *this;
}

fixed_decimal& operator/=(fixed_decimal const& that)
{//   division with rounding
      this->rep_ = rounded_division(this->rep_ * scale_, that.rep_);
      return *this;
}
```

The scale factor is the `N`th power of 10. I obtain this number as a compile time constant using a recursive template with a specialization to stop the recursion. This common template idiom is reproduced in Listing 4.

[Listing 4: computing the `N`th power of 10 using a template]

```
template <int N> struct exp10
{ enum { value = 10 * exp10<N-1>::value }; };
template <>      struct exp10<0>
{ enum { value = 1 }; };
```

Binary Fixed-point Template Class

In the binary fixed-point template class, a template parameter `N` gives the number of binary bits to the right of the radix point. Whole integer values are represented internally by multiplying the integer by a scale factor that is the `N`th power of 2; 2^7 or 128 gives about 2 decimal digits of precision. Thus the value 1.23 is represented approximately as 1.23*128 = 157. Note that 157/128 is 1.2265625, which rounds to 1.23 but is not exactly 1.23. The multiplication involved in scaling integers may be efficiently implemented as a left shift, which is faster than integer multiplication on most processors.

Addition and subtraction are implemented as ordinary integer operations on the scaled values, and always produce exact values.

Multiplication and division work the same as for decimal fixed-point numbers, except the scale factor can in principle be more efficiently applied as a left shift. The presented implementation uses the same scaled multiplication function as for the decimal fixed point type. This leaves it to the optimizer to discover the shortcut multiplication by left shifting.

An `int` has 31 bits of magnitude in VC7, so a `fixed_binary<7>` has 31 – 7 = 24 bits of magnitude, providing an integer range of +/-16,777,215. This may not be enough to count Bill Gates' fortune, but it's plenty for many applications.

Performance of the Library

As noted above, the fixed-point library produces computations more than twice as fast as corresponding computations using scalar double-precision floating-point arithmetic on the Pentium 4. But the results are more complicated than that. The performance of the fixed-point library is highly dependent on the unspecified internal architecture of the processor, as well as the quality of the compiler's optimizer. Potential users of this library would be well advised to test to be sure the library offers superior performance before replacing their floating point code.

On the Pentium 4, for instance, there is no difference in performance between 16-, 32-, and 64-bit integer fixed-point routines, presumably due to a 64-bit integer ALU, and perhaps due to the C++ rule that short expressions are widened to int. There was also no performance effect by changing fron long to long long intermediate results. The computation took longer with float than with double. Perhaps the FPU always works with double, and there was a conversion cost.

One correspondent with AMD hardware got only a 15 per cent gain in performance versus floating point (something I cannot verify without access to AMD hardware).

This implementation's performance owes much to the quality of optimization performed by the compiler. Visual C++ produced very respectable optimization. There is older, non-template fixed-point code on the net that posted discouraging performance marks in spite of using hand-coded assembly language functions for multiplication and division. The function call overhead overwhelmed any speed advantage from the assembly code.

Conclusion and Further Work

The fixed-point classes presented here provide useful basic functionality, but much more could be done. I have not yet investigated whether a template specialization for unsigned integral types would improve performance. There could be a function to convert among fixed-point types of differing precision. Multiplication (and division) could be distributed into two 32-bit multiplies (divides), to increase the range over which the product (quotient) can be computed without overflow and without 64-bit arithmetic. This would come at the cost of a second multiply (divide). Overflow could be handled, either by throwing an exception, or more intriguingly, by saturation; the replacement of an overflowed result by the largest representable value. I imagine a traits class to parameterize all this functionality.

I have used fixed-point arithmetic to improve the precision of computations for calibration and interpolation in analog to digital conversion, for computing latency in networks, and for collecting performance statistics in servers. Fixed-point math also has well known uses in graphics and digital signal processing. All these applications have common properties; a need for speed, a desire for accuracy, limited range of the computed values, and computations involving multiplication and division. For these applications there is no need to sacrifice speed for precision, because fixed-point arithmetic gives both.

Notes and References

[1] Rational numbers like 8/9 can be represented exactly as a separated numerator and denominator. Arithmetic using this representation requires repeated computation of the greatest common divisor, which is not an operation built into processors, or optimized by compilers. I haven't coded a rational number template implementation because I don't expect a performance advantage over floating-point.

[2] The values exactly represenable by a fixed-point type are spaced evenly through its range, and include each integer value. This isn't true for floating-point types. Fully half the exactly represented floating-point values are in the interval (-1,1). Exactly represented floating-point values become increasingly sparse as their magnitude increases.

[3] Boost operators, by Dave Abrahams and Jeremy Siek. http://www.boost.org/libs/utility/operators.htm

[4] www.guntheroth.com tells how to get an updated copy of this paper and the latest version of the fixed-point templates and support functions.