

Scouting Out Optimizations

by Stephen C. Dewhurst

Last time we examined “typeints”; a mechanism to implement extended-precision, compile time arithmetic using type manipulation in place of more traditional arithmetic operations. As a result, we were able to handle compile time arithmetic on integers many hundreds of bits in length.

As a parting shot, we had a quick look at a multiplication algorithm for typeints, but observed that the algorithm had quadratic compile time complexity. In this installment of Common Knowledge, we’ll investigate a number of metaprogramming techniques that can improve the observed performance of the algorithm by several orders of magnitude without affecting its asymptotic (big-O) complexity.

Iterative and Recursive Multiplication

We’ll use the simple shift-and-add algorithm for multiplication throughout this installment. This is why the asymptotic complexity results will not change as we apply various optimizations; the essential algorithm remains the same.

Recall that the shift-and-add approach to implementing multiplication entails examining the multiplier bit-by-bit. The product is computed by shifting the multiplicand left by the current bit position of the multiplier. If the multiplier bit is one, then the shifted multiplicand is added to the product. This procedure is perhaps more clearly described by example:

```
      1 1 0 1 // A == 13
x     1 0 0 1 // B == 9
-----
      1 1 0 1
     0 0 0 0
    0 0 0 0
   1 1 0 1
-----
  1 1 1 0 1 0 1 // A * B == 117
```

A standard iterative solution (substituting unsigned integers for typeints) might look something like the following:

```
unsigned multiply1( unsigned a, unsigned b ) {
    unsigned prod = 0;
    while( b ) {
```

```
        if( b & 1 )
            prod += a;
        a <<= 1;
        b >>= 1;
    }
    return prod;
}
```

However, since we'll be performing the computation at compile time with template metaprogramming, we don't have iteration available to us, so we have to use a recursive approach:¹

```
unsigned multiply2( unsigned a, unsigned b ) {
    if( b )
        if( b & 1 )
            return (multiply2( a, b>>1 )<<1) + a;
        else
            return multiply2( a, b>>1 )<<1;
    else
        return 0;
}
```

The recursive form of the algorithm essentially says, "First, recurse to multiply a by the higher-order bits of b, and shift the result. Then, if the lowest-order bit of b is one, add a to the previous result." The recursion terminates when b is zero.

Metaprogrammed Multiplication

We can use the implementation of this recursive algorithm to guide the coding of the metaprogrammed version. First, we must declare the primary template. However, we will not be using it, so we don't have to define it.

```
template <typename A, typename B>
struct Mul;    // no definition of primary
```

The first partial specialization handles the case where the lower-order bit of the typeint B is zero.² The implementation recursively instantiates the Mul template to compute the value of multiplying A by the higher-order bits of B, then shifts the result by appending an unqualified pointer modifier.

```
template <typename A, typename B>
struct Mul<A,B *>    // 0 digit, just shift
                    // A*B0 == (A*B)<<1
    { typedef typename Mul<A,B>::R *R; };
```

The second partial specialization handles the case where the lower-order bit of B is one. The implementation is essentially the same as in the previous case, but the original value of A is added to the

shifted result.³

```
template <typename A, typename B>
struct Mul<A,B *const> { // 1 digit, shift and add
    // A*B1 == ((A*B)<<1)+A
    typedef typename Mul<A,B>::R *Tmp;
    typedef typename Add<Tmp,A>::R R;
};
```

Penultimately, we terminate the recursion when B is zero.

```
template <typename A>
struct Mul<A,char> // A * 0 == 0
    { typedef char R; };
```

Finally, we must take care of a glitch. We special-case in the event that B is one by simply “returning” A as the result. The multiplication would succeed without this additional partial specialization, but would result in the typeint product’s having a leading zero. For convenience and efficiency, the typeint representation (described in the previous installment) does not permit leading zeros.

```
template <typename A>
struct Mul<A,char *const> // to avoid a char * typeint
    { typedef A R; }; // no leading zeros!
```

This metaprogrammed typeint multiplication is effective, in that it works:

```
typedef TypeInt<12345>::R N1;
typedef TypeInt<54321>::R N2;
typedef Mul<N1,N2>::R Product;
cout << Value<Product>::r << endl; // 670592745
```

However, as we noted at the close of the previous Common Knowledge, we may have to wait a while for the result. On a typical workstation/compiler combination, multiplication of two 128-bit typeints can require more than two hours!

Clumping Runs of Bits

Let’s see what optimization can be accomplished without actually changing the multiplication algorithm. The major source of (non-algorithmic) inefficiency would seem to be the repeated shifting and adding of the first (A) argument by the number of bits in the second (B) argument. One way to reduce the number of additions performed would be to perform a single shift and addition for a “run” of like bits, rather than a shift and addition for each bit of the multiplier.

1	1	0	1	1	1	1	1	0	0	0	0	1	0	0	0	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

4 Additions, 7 shifts

1	1	0	1	1	1	1	1	1	0	0	0	0	1	0	0	0	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

11 Additions, 19 shifts

Probabilistically, we'd expect some clumping of bits, so this approach should result in generally better performance. However, in the worst case—alternating zero and one bits—this approach is no better than the previous one, so the asymptotic complexity of the multiplication algorithm is unchanged.

Trading Places

An additional optimization would be to take advantage of the commutativity of multiplication, and make sure that the multiplier (B) is always the longer of the two operands, so as to have a better chance of encountering longer runs of like bits in B, and (more importantly) to minimize the number of bits in the multiplicand (A) to improve the efficiency of the remaining additions. Our optimized multiplication will therefore begin by making sure that the multiplier is the shorter of the two operands.

```
template <typename A, typename B>
struct Mulopt {
    // the second arg should be the larger of the two
    enum { alen = Len<A>::r, blen = Len<B>::r };
    typedef typename M2<(alen<=blen),A,B>::R R;
};
```

One of the implementations of the M2 template performs the actual switch. (We may call M2 an “implementation template” because it is not accessed directly by the user of our multiplication facility.)

```
// M2 switches the args, if necessary,
// so the longer arg is second arg.
template <bool, typename A, typename B>
struct M2 {
    typedef typename M3<A,B>::R R;
};

template <typename A, typename B>
struct M2<false,A,B> {
    typedef typename M3<B,A>::R R;
};
```

Note the desirability of the indirection from Mulopt to M2 to perform the operand swap. We could have implemented the swap directly within Mulopt, but in that case both branches of the conditional would have been instantiated:

```
template <typename A, typename B>
struct Mulopt { // Note: improper implementation!
    enum { alen = Len<A>::r, blen = Len<B>::r };
    typedef typename Select<
        (alen<=blen),
```

```
    typename M3<A,B>::R, // this is always instantiated..
    typename M3<B,A>::R // and so is this!
>::R R;
};
```

The result would be correct, but we would have more than doubled the compile time, since we'd have performed the multiplication twice! Since the entire purpose of this exercise is to reduce compile time, this approach is clearly not appropriate, and the indirect instantiation of the `M2` template is required in order to prune the unused instantiation path.

Specializations of `Mulopt` also handle the trivial cases where one or both operands are zero. This has the effect of simplifying the rest of the implementation.

```
// three specializations for zero args
template <typename A>
struct Mulopt<A,char> {
    typedef char R; // A * 0 == 0
};

template <typename B>
struct Mulopt<char,B> {
    typedef char R; // 0 * B == 0
};

template <>
struct Mulopt<char,char> {
    typedef char R; // 0 * 0 == 0
};
```

Scouts

A metaprogramming technique that seems to have general applicability is the use of scouts. A “scout” is simply a small section of code that performs a compile time lookahead in order to determine how to proceed. The term is meant to evoke the practice of sending a single individual ahead of a larger body of explorers in order to determine the lay of the land. A runtime analogy might be a parser’s use of one or more symbols of lookahead in order to determine the direction of the parse.

In our case, the scouting report will be fairly straightforward: do we have a run of zeros or ones ahead, and how many do we have? The implementation template `M3` uses the scouting report to decide how to proceed.

```
template <typename A, typename B>
struct M3 {
    enum {
```

```
        ones = Count1s<B>::r,  
        zeros = Count0s<B>::r  
};  
typedef typename M4<zeros,ones,A,B>::R R;  
};
```

Note that the value of at least one of the enumerators `ones` and `zeros` will be zero. The scouts themselves are trivial (recall that a `*` modifier represents a zero bit, and a `* const` modifier a one bit):

```
template <typename T>  
struct Count0s {  
    enum { r = 0 };  
};  
  
template <typename T>  
struct Count0s<T *> {  
    enum { r = Count0s<T>::r + 1 };  
};  
  
template <typename T>  
struct Count1s {  
    enum { r = 0 };  
};  
  
template <typename T>  
struct Count1s<T *const> {  
    enum { r = Count1s<T>::r + 1 };  
};
```

Once `M3` has received the scouting report, it knows how to proceed by instantiating the appropriate version of the implementation template `M4`. All the possible cases are handled by partial specializations of `M4`, but we still have to declare the primary:

```
template <int zeros, int ones, typename A, typename B>  
struct M4;
```

The end condition of the recursion occurs when the multiplier (`B`) has been stripped of all its bits:

```
template <typename A, typename B>  
struct M4<0,0,A,B> { // nothing left of B  
    // specializations of Mulopt assure us no  
    // zero args so we can return 1 here
```

```
typedef char * const R;
};
```

If the scouts have reported a run of zero bits, all we have to do is shift the run off of B, multiply A by the result, and shift A by the length of the run.

```
template <int n, typename A, typename B>
struct M4<n,0,A,B> { // B has run of n 0s
    typedef typename RShift<B,n>::R T1; // B/2**n
    typedef typename Mulopt<A,T1>::R T2; // A*(B/2**n)
    typedef typename LShift<T2,n>::R R; // (A*(B/2**n))*2**n
};
```

The expression $(A * (B/2^n)) * 2^n$ is equivalent to $A * B$ if B's lower n bits are zero. Note that the multiplication and division are accomplished with the left and right shift operators for typeints described in the previous installment. This is a marginal improvement over processing each zero bit individually, but we haven't yet avoided any additions.

Note also that the recursive instantiation to produce the result $A * (B/2^n)$ is made to `Mulopt` rather than the implementation template `M3`. The reason is that, after removing the run of n bits from B, the length of B may now be less than that of A, and we'd like to reapply the swap optimization described earlier.

We get most of the improvement in handling runs of ones. If we have a run of one bits, we do the same as with a run of zero bits, but we have to add in the result of A times that run of one bits from B.

```
template <int n, typename A, typename B>
struct M4<0,n,A,B> { // B has run of n 1s
    typedef typename RShift<B,n>::R T1;
    typedef typename Mulopt<A,T1>::R T2;
    typedef typename LShift<T2,n>::R T3;
    typedef typename LShift<A,n>::R Ax2n;
    typedef typename Sub<Ax2n,A>::R Ax2n_1;
    typedef typename Add<T3,Ax2n_1>::R R;
};
```

The expression $((A * B / 2^n) * 2^n) + (A * (2^n - 1))$ is equivalent to $A * B$ if B's lower n bits are one. And $A * (2^n - 1)$ is equivalent to $A * 2^n - A$ which is equivalent to $(A \ll n) - A$. Translated into a more conventional calculation, the typedefs in this specialization of M4 are equivalent to the following:

```
T1 = B >> n;
T2 = A * T1;
T3 = T2 << n;
Ax2n = A << n;
Ax2n1_1 = Ax2n - A;
R = T3 + Ax2n1_1;
```

Limited Subtraction

One final piece is required for the implementation of these optimizations; the subtraction algorithm employed by the specialization of `M4`, above. For our purposes, we'll implement an unsigned subtraction that assumes that the first argument is greater than the second. Most of the implementation is straightforward:

```
template <typename A, typename B>
struct Sub;

template <typename A, typename B>
struct Sub<A *const, B *const> { // A1-B1 == (A-B)0
    typedef typename Sub<A, B>::R *R;
};

template <typename A, typename B>
struct Sub<A *const, B *> { // A1-B0 == (A-B)1
    typedef typename Sub<A, B>::R *const R;
};

template <typename A, typename B>
struct Sub<A *, B *> { // A0-B0 == (A-B)0
    typedef typename Sub<A, B>::R *R;
};

template <typename A>
struct Sub<A, char> { // A-0 == A
    typedef A R;
};

template <>
struct Sub<char, char> { // 0-0 == 0
    typedef char R;
};
```

The interesting case occurs when a one bit in `B` is subtracted from a zero bit in `A`, and it is necessary to “borrow” from the higher-order bits of `A`:

```
template <typename A, typename B>
struct Sub<A *, B *const> { // A0-B1 == ((A-1)-B)1
```

```
// Need to borrow.
typedef typename Borrow<A>::R T1;
typedef typename Sub<T1,B>::R *const R;
};
```

Borrow scans back to find a one bit, zeros it, and sets all the preceding zero bits to one.

```
template <typename T>
struct Borrow;

template <typename T>
struct Borrow<T *const> {
    // make sure there's no leading 0
    typedef typename Select<IsPtr<T>::r,T *,T>::R R;
};

template <typename T>
struct Borrow<T *> {
    typedef typename Borrow<T>::R *const R;
};
```

Results

With these optimizations in place, anecdotal evidence indicates the performance of the `Mulopt` implementation of the multiplication algorithm is typically 2-3 times faster than the original `Mul` implementation. In special cases in which there are large runs of like bits, `Mulopt` can outperform `Mul` by several orders of magnitude. However, it's not often that extended precision compile time multiplication will be of practical use.⁴

However, the techniques used in implementing the optimizations are of general, practical use, and can be commonly employed in template metaprogramming.

¹ See Andrei Alexandrescu's *Modern C++ Design*, Addison-Wesley 2001, p. 54 for an explanation as to why recursion is necessary.

² Throughout this article, we'll employ the usual terminology of unsigned binary arithmetic to refer to the analogous typeint concepts. For instance, we describe a pointer modifier that is part of a typeint as a "bit," a const-qualified pointer modifier as a "one bit," an unqualified pointer modifier as a "zero bit," and the type `char` as the numeric value zero.

³ We described the implementation of the `Add` template in the previous installment. The source code for the complete typeint implementation is available at <http://www.semantics.org/code.html>.

⁴ However, shifting and bitwise operations on extended precision integers do seem to be more commonly useful. Fortunately, these operations are easy to implement with linear compile time complexity.