

N-Ary Nibbling

by Stephen C. Dewhurst

Last time, we looked at improving performance of typeint multiplication through application of a couple of simple template metaprogramming techniques. In this installment, we'll take up the problem of compiler limitations on template instantiation depth.

Cursing and Recursing

Many template metaprogramming techniques rely heavily on recursion. For example, in the February installment of this column¹, we implemented a left-shift operation on typeints with a simple recursion.

```
template <typename T, int n>
struct LShift {
    typedef typename LShift<T,n-1>::R *R; // recurse
};

template <typename T>
struct LShift<T,0> {
    typedef T R;
};
```

This is the metaprogramming analog of the following recursive runtime implementation:

```
unsigned lshift( unsigned t, int n ) {
    if( n )
        return lshift( t, n-1 ) << 1;
    else
        return t;
}
```

This approach works well for small values of *n*, but fails when *n* is large.

```
typedef TypeInt<42>::R N;
typedef LShift<N,5>::R NLS5; // works...
typedef LShift<N,1000>::R NLS1000; // likely error!
```

The problem is that the second instantiation of `LShift` above is likely to cause the compiler to exceed its

maximum template instantiation depth. That is, instantiation of `LShift` for the values `N` and `1000` will (recursively) cause an instantiation of `LShift` for the values `N` and `999`, which will in turn cause the instantiation of `LShift` with the values `N` and `998`, and so on, and on and on. Compilers vary widely in their ability to handle deeply nested template instantiations. In Annex B of the C++ standard, we find a suggested minimum instantiation depth capability of only 17. However, most compilers seem to be able to handle instantiation depths of at least 60, and others can manage more than 1000, depending on available resources.

We can improve this miserable situation by recognizing that there's no need to recurse in a single, deep thrust as we have above. We can “nibble” a fixed, modest recursion depth, back out, and nibble again.

```
const int nbl = 10; // depth of a nibble
//...
template <typename T, int n>
struct LShiftNbl {
    typedef typename LShiftImpl<(n>=nbl), T, n>::R R;
};
```

Here we select the recursion approach based on the shift length. If the shift length is at least the size of a nibble, we nibble and try again. Otherwise we employ the original, deep recursion because the depth of the recursion is reasonably bounded.

```
template <bool, typename T, int n>
struct LShiftImpl { // nibble...
    typedef typename
        LShiftImpl<(n-nbl)>=nbl), T, n-nbl>::R *****R;
};

template <typename T, int n>
struct LShiftImpl<false, T, n> { // finish up any crumbs
    typedef typename LShiftImpl<false, T, n-1>::R *R;
};

template <typename T>
struct LShiftImpl<false, T, 0> {
    typedef T R;
};
```

With this approach, we can process left shifts an order of magnitude greater in length than we could previously.

```
typedef LShiftNbl<N, 1000>::R NLS1000; // works!
```

Systematic Base N Nibbling

The nibble approach is effective, but ad hoc. It would be preferable to employ it uniformly throughout

the design of the typeint implementation. Let's do that.

The original typeint implementation uses a binary representation; a const-qualified pointer modifier represents a one bit, and an unqualified pointer modifier a zero bit. For example, we can represent the binary integer 1011 (decimal 11) as `char *const * *const *const`. We could improve matters somewhat by moving to a base three or base four implementation. For base four we could represent the digit 2 as a volatile-qualified pointer modifier, and 3 as a const volatile-qualified pointer modifier. For example, the base four integer 1203 (decimal 99) could be represented as `char *const *volatile * *const volatile`.

Rather than move the implementation from binary to ternary or base 4, let's move it from binary to base 2^L . We'll abandon use of the pointer modifier in preference to the array modifier. For example, we can represent the integer `0xBEEF` (decimal 48879) as the type `char [11][14][14][15]` if `L` is chosen to be 4 (base 16), or `char [11][59][47]` if `L` is chosen to be 6 (base 64).

We must, however, make a couple of adjustments to this simple transformation. First, in order to avoid the potential of an illegal zero array bound, we'll add one to each "digit" in the representation. Therefore, the representation of decimal 48879 is rendered as `char [12][15][15][16]` if `L` is 4, and as `char [12][60][48]` if `L` is 6.

Second, it is likely that the compiler will impose a maximum object size, which in turn implies that the `sizeof` operator applied to a type must be similarly limited.² In other words, it is likely that the maximum size of a type is the same as that of an object. In Annex B, the C++ standard suggests that a compiler allow an object of at least size 262144 bytes (256x1024, or 2^{18}). This limit can be exceeded quite quickly by a multidimensional array; for instance, the integer `0xDEADBEEF` rendered as `char [14][15][11][14][12][15][15][16]` is well over this limit.

We can circumvent this difficulty by the use of *pointers* to arrays, rather than arrays. For example, we can render 48879 as `char (*(*(* (* (* [0xF +1]) [0xE +1]) [0xE +1]) [0xB +1])`; a pointer to an array of 16 pointers to arrays of 15 pointers to arrays of 15 pointers to arrays of 12 pointers to `char`.

```
typedef char (*( *(* (* (*Beef) [0xF +1]) [0xE +1]) [0xE +1]) [0xB +1]);
Beef beef;
```

Say what you will about its complexity, but `sizeof(Beef)` is the size of a simple pointer. It's still necessary to show some caution to ensure that the size of any subsidiary type is still within the compiler's object size constraints. For example, `sizeof(*beef)` or `sizeof(*(*beef)[0])` should still be less than 2^{18} . Effectively, this means that an array bound in the representation must be no greater than 2^{18} divided by the `sizeof` a pointer to data, or (probably) 2^{16} . Since we must augment the bound by one, a safe maximum value for `L` might be 15, allowing us to perform arithmetic in base 32768. However, the implementation will work for any value of `L` in the range of 1 to 15 (or whatever the local maximum might be).

N-Ary Typeints

Of course, no rational person would want to construct such a type by hand (I'm not always completely rational). Fortunately, it's straightforward to modify our earlier binary typeint implementation to perform the construction for us:

```
template <int n>
```

```
struct TypeInt {
    typedef typename TypeInt<(n>>L)>::R (*R) [(n&MASK(L))+1];
};

template <>
struct TypeInt<0> {
    typedef char R;
};
```

Our earlier implementation recursively converted the value $n \gg 1$ to a binary typeint, then appended the appropriate pointer modifier depending on whether the low-order bit was a one or zero. In this n-ary implementation, we're using base 2^L , not binary (unless $L == 1$), so we recursively find the typeint representation for $n \gg L$, then append a pointer to array modifier to the result. The array bound is the lower-order L bits of n , incremented by one (again, to avoid the possibility of a zero array bound). We've effectively applied the "nibbling" technique described above, where the size of the nibble is L .

The conversion from an n-ary typeint to an integer (assuming the typeint can fit in a predefined integer) is essentially the reverse operation:

```
template <typename T>
struct Value {
    enum { r = 0 };
};

template <typename T, int bound>
struct Value<T (*) [bound]> {
    enum { r = (Value<T>::r << L) + bound-1 };
};
```

We strip the outermost pointer-to-array from the typeint and recursively convert the remaining typeint to an integer. We then shift that result left by L bits, and add in the adjusted (that is, we subtract the 1 we added when constructing the typeint) lower L -bit digit. Again, this is identical to the binary typeint implementation, but we are proceeding in L -bit nibbles, with concomitant savings in recursion depth. In general, we will be able to handle integers that are on the order of L times larger than we could previously. Note that we employ two different, effective conceptualizations of the typeint structure. Often it's convenient to consider an n-ary typeint as a simple sequence of digits³ in base 2^L , and at other times as a binary representation whose bits are manipulated in groups of L . Either conceptualization is accurate, so we will use both in the description below.

Shifting N-Ary Typeints

Let's consider performing a left shift.⁴ In implementing the shift operations, it's convenient to view the typeint as a sequence of groups of L bits.

If the length of the shift, n , is less than L , then we must simply shift left by n bits the values of the array

bounds that comprise the typeint. Any bits that “fall off the end” of one bound will be ored on to the lower n bit positions of the shifted result of the next bound.

```
template <typename T, int n, int lostbits>
struct LShiftShort {
    typedef T R;
};

template <typename T, int n, int bound, int lostbits>
struct LShiftShort<T(*) [bound],n,lostbits> {
    enum {
        mylostbits = LSLOST(bound-1,n),
        newb = (BOUND((bound-1)<<n) // limit result to L bits!
                | lostbits) // add in bits lost from prev digit
                + 1 // avoid zero bound
    };
    // If mylostbits is non-zero, and this is the last
    // modifier (prev bound==0), then we have to add a
    // new last modifier with the lost bits (otherwise we'd
    // lose the high-order digit).
    typedef typename LShiftShort<T,n,mylostbits>::R Temp;
    typedef Temp (*RR) [newb];
    typedef typename Select<
        (mylostbits && (IsAry<typename
            Deref<T>::R>::bound==0)),
        RR (*) [mylostbits+1],
        RR
    >::R R;
};
```

The facilities BOUND and LSLOST are (I’m sorry to say) implemented as preprocessor macros, and are typical bit-twiddling implementations.⁵ BOUND is used to truncate a shifted result to L bits. LSLOST determines what bits will be lost in shifting an L -bit quantity left by n bits:

```
#define MASK( n ) ((1<<(n))-1)
#define BOUND( b ) ((b) & MASK(L))
#define END( b, n ) ((b) & (MASK(n)<<(L-(n))))
#define LSLOST( b, n ) (END(b,n)>>(L-(n)))
```

In order to handle shift lengths that are longer than L , we first shift the typeint by $n\%L$ bits, then append

n/L “zero” digits to the end of the typeint:

```
template <typename T, int n>
struct LShift {
    typedef typename LShiftShort<T,n%L,0>::R Temp;
    typedef typename AddPtrAryN<Temp,n/L>::R R;
};
```

The AddPtrAryN facility is just one of a number of simple ad hoc utilities that make it easier to work with the n-ary representation of typeints.

```
template <typename T, int n>
struct AddPtrAryN {
    // append “zero” digit
    typedef typename AddPtrAryN<T,n-1>::R (*R) [1];
};
```

```
template <typename T>
struct AddPtrAryN<T,0> {
    typedef T R;
};
```

Bitwise Operations

The implementations of the various bitwise operations on n-ary typeints are similar. Let’s examine the implementation of bitwise or as a representative example. The primary template is declared but not defined. All legal cases are covered by specializations of the primary, and any ill-formed typeints will cause attempted instantiation of the primary and a compile time error.

```
template <typename T1, typename T2>
struct Or; // no definition
```

A complete specialization takes care of the case where both arguments are zero:

```
template <>
struct Or<char,char> { // 0 | 0 == 0
    typedef char R;
};
```

The cases in which one argument is zero and the other non-zero are also straightforward:

```
template <typename T1, int b1>
struct Or<T1(*) [b1],char> {
    typedef T1 (*R) [b1]; // T1 | 0 == T1
};
```

```
template <typename T2, int b2>
struct Or<char, T2 (*) [b2]> {
    typedef T2 (*R) [b2]; // 0 | T2 == T2
};
```

The interesting case occurs when both arguments are non-zero:

```
template <typename T1, int b1, typename T2, int b2>
struct Or<T1 (*) [b1], T2 (*) [b2]> {
    typedef typename Or<T1, T2>::R Temp;
    enum { newbound = ((b1-1) | (b2-1)) + 1 };
    typedef Temp (*R) [newbound];
};
```

First we recurse to calculate the result of or-ing the higher-level digits of the two typeints (the calculation of `Temp`, above). We then calculate the value of the low-order digit by or-ing the values of the outermost array bounds of the typeint arguments. Note that we had to adjust the bounds before or-ing their values to take into account the earlier addition of one to each bound, and increment the result by one to avoid the possibility of a zero array bound. Finally, we append a pointer-to-array modifier with the new bound to the value previously calculated (`Temp`).

The logic of this implementation is essentially the same as that of the binary typeint implementation, but we are performing the calculations in nibbles of length `L`. As a result, the depth of recursive template instantiation required is reduced by a factor of `L`.

Results

Experience indicates that systematic use of nibbling to reduce recursive template instantiation depth in the typeint implementation permits manipulation of significantly larger typeints than the simple binary implementation. Platforms that formerly were able to handle only typeints of approximately 60 bits are able to handle typeints of over 500 bits. Platforms that could previously handle typeints of approximately 500 bits are able to handle typeints of over 5000 bits.

¹ S.C. Dewhurst, Typeints, CUJ online experts 21(2), February 2003.

² Thanks to Andrei Alexandrescu for making me see this as right and proper, rather than a bug in a compiler or the standard.

³ The use of the term “digit” in anything but a base 10 number system is a bit of an oxymoron, but the terminology is commonly used in binary arithmetic, as in “binary digit,” so I’m re-employing it here.

⁴ The implementation of right shift is similar. The full implementation of n-ary typeints is available from the author’s web site at <http://www.semantics.org/code.html>.

⁵ This is a common problem in programming with templates. An integer argument to a class template must be a constant-expression. Ordinarily, we prefer to use inline functions in preference to preprocessor macros, but the result of calling an inline function—even though it may be determinable at compile time—is not a constant expression, and cannot be used as an argument in template instantiation.