

Typeints

by Stephen C. Dewhurst

In a recent series of articles, we designed and implemented a `typeof` facility for C++.¹ One part of the implementation entailed a compile time transformation of an expression's type into a structured integer, that was later decoded to regenerate the original type. One problem we faced was the size limitation imposed on the integer encoding due to the necessity of performing compile time arithmetic on integers of modest precision (32 or 64 bits), while encodings of many "reasonable" types might require integers hundreds of bits in length. ("Unreasonable" types could require integers of unbounded size.)

Since traditional data abstraction techniques are not available at compile time, we were unable to implement an extended precision integral type; we were limited to those operations that could be employed in integer constant-expressions. Our approach was to implement compile time arithmetic operations on collections of predefined integers. For example, to left shift a quad-precision integer (composed of four integer constant-expressions `c1-c4`) we would write code similar to the following:

```
typedef ShiftLeft<c4, c3, c2, c1, ModLen> SL;
enum {
    code1 = SL::code1,
    code2 = SL::code2,
    code3 = SL::code3,
    code4 = SL::code4
};
```

This is effective, but ungainly. We are also forced to choose the maximum precision of our result a priori (four, in this case), rather than let the complexity of the encoding adjust the precision of the result implicitly.

As we noted above, compilers have strengths and weakness that correspond to the requirements imposed on them by the language they are designed to translate. A C++ compiler has need of only modest compile time arithmetic ability, but requires extensive ability to manipulate complex types. In fact, most well-designed compilers have a notion of a "type algebra," in which types are manipulated at compile time with various type-algebraic operations. For example, as the declaration of a name is parsed, the compiler will compose the name's type from its declaration-specifiers and declarators. When the name is manipulated in expressions, its type will be manipulated in sync by other type-algebraic operations.

This implies that it might be profitable to represent integers as types, and to map compile time arithmetic operations onto type manipulation operations, so as to leverage the compiler's extensive abilities in that area. We should then be able to perform extended precision compile time arithmetic on this integer/type abstraction. If necessary, portions of the extended precision results could be mapped back to the corresponding predefined integral type.

Mapping Integers to Types

Let's define a simple mapping from unsigned binary integers to C++ types by representing a zero bit as a pointer modifier, and a one bit as a const-qualified pointer modifier. The pointer's base type doesn't matter (much), so we'll choose `char` for that purpose. The unmodified base type will represent the value zero. In order to support variable length typeints, we'll disallow leading zero bits (that is, unqualified pointer modifiers) in the type representation.

For example, with this simple encoding we can represent the unsigned binary integer 1011100 (decimal 92) as the type `char *const * *const *const *const * *`. Let's call this type-analog of an integer a "typeint."

It's fairly easy to convert a small integer into a typeint:²

```
template <int n>
struct TypeInt {
    typedef typename Select<
        n&1,
        typename TypeInt<(n>>1)>::R *const,
        typename TypeInt<(n>>1)>::R *
    >::R R;
};

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

The primary `TypeInt` template recursively determines the encoding for high-order bits of the integer `n`, then appends a `* const` if the low-order bit is one, or a `*` if the low-order bit is zero. The recursion terminates when the complete specialization of `TypeInt` is instantiated when `n` is zero.

Converting a typeint to an integer is also straightforward, provided the integral value represented by the typeint can fit into a predefined integer:

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

template <typename T>
struct Value<T *>
{ enum { r = Value<T>::r*2 }; };

template <typename T>
struct Value<T * const>
```

```
{ enum { r = Value<T>::r*2 + 1 }; };
```

In this case, the primary template is used to terminate the recursion when all the “bits” (pointer modifiers) have been stripped from the typeint. The partial specializations recursively calculate the integer equivalent of the dereferenced typeint, and then append a 0 or 1 bit based on the outermost pointer modifier.

```
cout << Value< TypeInt<92>::R >::r << endl; // 92
```

Shifting Typeints

Left shift will zero-fill, so we have only to append the correct number of zeros:

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

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

However, our typeint representation does not permit leading zeros, so we must special case for the possibility of shifting a zero (char) typeint:

```
template <int n>
struct LShift<char,n>
    { typedef char R; };
```

Finally, we must provide a complete specialization in order to disambiguate the case where a zero typeint is left-shifted zero bits. Otherwise, there would be an ambiguity between the partial specializations.

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

The right shift operation always zero-fills in this implementation. Since there are no leading zeros in this implementation, this means we simply dereference the typeint the appropriate number of times:³

```
template <typename T, int n>
struct RShift {
    typedef typename Deref<typename RShift<T,n-1>::R>::R R;
};

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

Bitwise Operations

Let's look at an implementation of bitwise or. Addition and the other bitwise operations are similar. In this implementation, we'll employ a compile time switch to separate out the cases when one or both of the arguments is zero:

```
template <typename A, typename B>
struct Or {
    enum { switchval = IsPtr<A>::r*2 + IsPtr<B>::r };
    typedef typename OrImpl<switchval,A,B>::R R;
};
```

Since a zero typeint value is represented by an unqualified `char` base type, we can construct a switch-expression based on whether the arguments to the bitwise or are both non-zero (they're both pointers; the switch value is 3), only the first argument is non-zero (2), only the second argument is zero (1), or both arguments are zero (0). As one might expect, the cases with one or two zero arguments are trivial:

```
template <typename A, typename B>
struct OrImpl<0,A,B> {
    typedef char R; // 0 | 0 == 0
};
template <typename A, typename B>
struct OrImpl<2,A,B> {
    typedef A R; // A | 0 == A
};
template <typename A, typename B>
struct OrImpl<1,A,B> {
    typedef B R; // 0 | B == B
};
```

When both arguments are non-zero, things are a bit more interesting:

```
template <typename A, typename B>
struct OrImpl<3,A,B> {
    typedef typename Deref<A>::R DA;
    typedef typename Deref<B>::R DB;
    typedef typename Or<DA,DB>::R Temp;
    typedef
    typename Select<
        IsConst<A>::r || IsConst<B>::r, // either A or B has a
1 bit
        Temp * const,
        Temp *
```

```
>::R R;  
};
```

First, we dereference the two arguments to remove the low-order bit of each, and recursively calculate the result of bitwise or-ing the two shortened typeints. We then append the or-ed value of the low-order bits we removed earlier.

This use of a compile time switch may seem needlessly complex. Another approach might be to use exhaustive specialization instead. There are nine cases to consider, as the table below illustrates.

First Argument	Second Argument
char	char
char	B *
char	B * const
A *	char
A *	B *
A *	B * const
A * const	char
A * const	B *
A * const	B * const

The primary and eight partial specializations follow those listed in the table, above.

```
template <typename A, typename B>  
struct Or { typedef char R; };  
  
template <typename A, typename B>  
struct Or<A *,B *> { typedef typename Or<A,B>::R *R; };  
  
template <typename A, typename B>  
struct Or<A *const,B *> { typedef typename Or<A,B>::R *const R;  
};  
// 5 more cases...  
template <typename A, typename B>  
struct Or<A *const,B> { typedef A *const R; };
```

In the case of the implementation of bitwise or, it's a tossup as to which implementation mechanism is preferable. The implementation of bitwise and can merge some cases and is, therefore, easier to implement by exhaustive specialization.

```
template <typename A, typename B>
```

```
struct And // at least one of A or B is 0
{ typedef char R; };

template <typename A, typename B>
struct And<A *,B *>
{ typedef typename And<A,B>::R *R; };

template <typename A, typename B>
struct And<A *const,B *>
{ typedef typename And<A,B>::R *R; };

template <typename A, typename B>
struct And<A *,B *const>
{ typedef typename And<A,B>::R *R; };

template <typename A, typename B>
struct And<A *const,B *const>
{ typedef typename And<A,B>::R *const R; };
```

Uses and Limitations

The use of typeints for extended compile time arithmetic has some advantages over our earlier ad hoc mechanism. The original mechanism was wordy and fixed to a given precision a priori:

```
typedef ShiftLeft<c4,c3,c2,c1,ModLen> SL;
enum {
    code1 = SL::code1,
    code2 = SL::code2,
    code3 = SL::code3,
    code4 = SL::code4
};
```

The use of a typeint is sparer, and we do not have to specify the precision of the result in advance. The typeint will expand its precision as needed, up to the capacity of the compiler to manipulate it:

```
typedef LShift<aCode, ModLen>::R Code;
```

Unfortunately, that's where a problem lies. The typeint implementation is highly recursive, and many compilers cannot perform recursive template instantiation beyond a very modest depth. This effectively limits the maximum length, in bits, of a typeint to be conformant with the maximum instantiation depth of the compiler. While the C++ standard suggests a minimum instantiation depth of only 17 levels (in Annex B), most compilers seem to be able to manage at least 60, while others have compiler options that

will allow instantiation depth to be as high as 600-700, and still others will instantiate until resource exhaustion, typically giving results in the low thousands. We'll examine approaches to dealing with the problem of recursive template instantiation depth limitations in a future column.

Complex Multiplication

Before we close, let's have a too-quick look at an implementation for typeint multiplication. We'll implement it using the traditional shift-and-add approach:

```

    1 1 0 1 // A
  x 1 0 0 1 // B
  -----
    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

```

The representation of the above calculation with typeints is essentially the same, but is somewhat less readable:

```

                                char *const *const * *const
                                x                                char *const * * *const
                                -----
                                char *const *const * *const
                                char
                                char
                                char *const *const * *const * * *
                                -----
                                char *const *const *const * *const * *const

```

A straightforward implementation of the shift and add multiplication algorithm is pleasingly spare:

```

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

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

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; };

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;
};

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

Unfortunately, this algorithm has quadratic (compile time!) complexity rather than the linear complexity of the implementations of the other operators. As a result, on a typical workstation there is a noticeable compilation delay when multiplying typeints of 35 digits, and multiplication of 200-digit operands implies a lunch break at a minimum.

Next Time

In the next installment, we'll look at the multiplication implementation in more detail, and examine a number of template metaprogramming techniques that dramatically improve its performance. In the installment that follows the next, we'll examine techniques for dealing with template instantiation depth limitations.

¹ Dewhurst, S.C. A Bit-Wise Typeof Operator, Parts 1-3. C/C++ Users Journal v. 20, no. 8, 10, and 12 (August, October, and December 2002).

² The full implementation of this preliminary typeid facility is available on the author's website: <http://www.semantics.org/code.html>. The reader may find in the same location a reimplement of the typeid facility that employs typeid for extended precision compile time arithmetic. The `Select` template is borrowed (in modified form) from Andrei Alexandrescu's Loki library. It performs a compile time if-statement, selecting its second argument if the first argument is true, and its third argument if the first argument is false.

³ Description of utilities like `Select`, `IsPtr` and `Deref` are omitted for reasons of space. Their implementations are available as described in the previous note.