

A Bit-Wise Typeof Operator, Part 2

by Stephen C. Dewhurst

In the last installment of this column, we looked at an approach for producing a Gödel numbering for the C++ type system. That is, we described a mapping from the set of C++ types to the set of integers, such that the mapping is structured and invertible. The goal is to implement an effective, portable `typeof` facility by conducting a “round-trip” translation of a type to an integer and back to a type again.¹ Step one employs function template argument deduction to instantiate a “Gödel number” template for the type of an expression. Step two extracts the Gödel number from the template as an enumerator defined within the instantiated template. Step three decodes the Gödel number to regenerate the type of the expression. Performing this sequence of operations at compile time allows us to determine the type of an expression: a `typeof` operation. In this installment, which is part two of three, we’ll implement the code to extract an expression’s type and generate an integer from it.

At the conclusion of part one, we described two outstanding implementation problems. First, we have to be able to construct and manipulate integers of unbounded size. Second, we have to be able to perform rather involved recursive bit twiddling on these large integers at compile time. That’s just what we’ll do for the remainder of this column. Be warned that this column is very heavy on implementation and contains a lot of code but, even so, there isn’t space to include the entire implementation.²

Multipart Integer Operations

Creating a usable integer of arbitrary precision is not difficult if one has access to all of C++’s facilities for constructing abstract data types. However, we are limited by the need to manipulate our large integers at compile time. This means that we are restricted to the use of integer constant-expressions, which in turn means that we must use only predefined integral types.

Well, all is not lost. If we can’t create any new types for compile time arithmetic, we can instead create new operations on the existing types. Let’s look at a “left-shift” operation on an unsigned integer type.

```
typedef unsigned short Code;
const Code CodeLen
    = std::numeric_limits<Code>::digits;
#define MASK( n ) \
    ((1<<((n)-1))-1) | (1<<((n)-1))
const Code CodeMax = MASK(CodeLen);
#define BOUND( n ) ((n) & MASK(CodeLen))
//...
template <Code c, Code n>
```

```
struct ShiftLeft1 {
    enum {
        basemask = MASK(n),
        leftendmask
            = BOUND(basemask << CodeLen-n),
        lostbits
            = BOUND((c & leftendmask) >> CodeLen-n),
        r1 = BOUND(c << n)
    };
};
```

The `ShiftLeft1` template simply shifts its first argument left by the number of bits specified by its second argument. The shifted result is available as the nested enumerator `r1`. Any bits that fall off the left end of the first argument as the result of the shift are available in the enumerator `lostbits`. For example, `ShiftLeft1<0xFFFF, 4>::r1` is `0xFFF0`, and `ShiftLeft1<0xFFFF, 4>::lostbits` is `0x000F`. Note the need to `BOUND` the shifted results by zeroing out bits in positions greater than `CodeLen`. Otherwise, depending on the underlying integral type the compiler uses to represent the enumerator, the bits that should have “fallen off the end” of the shifted `Code` may be retained in the enumerator, giving incorrect results.³

The ability to perform a compile-time shift of a single integral value is not so useful in itself, but we can use `ShiftLeft1` to implement a left shift operation on a pair of `Codes`.

```
template <Code c2, Code c1, Code n>
struct ShiftLeft2 {
    enum {
        r1 = ShiftLeft1<c1,n>::r1,
        r2 = ShiftLeft1<c2,n>::r1
            | ShiftLeft1<c1,n>::lostbits,
        lostbits = ShiftLeft1<c2,n>::lostbits
    };
};
```

The result of shifting the multipart integer composed of `c2` and `c1` left by `n` bits is available in the enumerators `r2` and `r1`; we can consider `r2` and `r1` as the double-precision result of performing a left shift of `n` bits on the double-precision argument composed of `c2` and `c1`. We can continue in this fashion to produce a left shift operation on an unbounded number of `Codes`. Inasmuch as I have a life, I decided to stop at four, though a more reasonable lower limit for a production `typeof` operator might be twelve or more.

```
template <Code c4, Code c3, Code c2, Code c1, Code n>
struct ShiftLeft4 {
    enum {
        r1 = ShiftLeft3<c3,c2,c1,n>::r1,
```

Dewhurst — A Bit-Wise Typeof Operator, Part 2

```
    r2 = ShiftLeft3<c3,c2,c1,n>::r2,
    r3 = ShiftLeft3<c3,c2,c1,n>::r3,
    r4 = ShiftLeft1<c4,n>::r1
        | ShiftLeft3<c3,c2,c1,n>::lostbits,
    lostbits = ShiftLeft1<c4,n>::lostbits
};
};
```

This mechanism works well if the length of the shift is less than the number of bits in `Code`, but fails for longer shift amounts. Let's put a layer over our simple left shift operation to take long shifts into account.

```
template <Code c, Code n>
struct ShiftLeftLong1 {
    enum {
        r1 = (n>CodeLen) ? 0
            : ShiftLeft1<c,n>::r1
    };
};

template <Code c2, Code c1, Code n>
struct ShiftLeftLong2 {
    enum {
        r1 = ShiftLeftLong1<c1,n>::r1,
        r2 = (n>CodeLen)
            ? ShiftLeftLong1<c1,n-CodeLen>::r1
            : ShiftLeft2<c2,c1,n>::r2
    };
};

//...

template <Code c4, Code c3, Code c2, Code c1, Code n>
struct ShiftLeftLong4 {
    enum {
        r1 = ShiftLeftLong3<c3,c2,c1,n>::r1,
        r2 = ShiftLeftLong3<c3,c2,c1,n>::r2,
        r3 = ShiftLeftLong3<c3,c2,c1,n>::r3,
        r4 = (n>CodeLen)
            ? ShiftLeftLong3<c3,c2,c1,n-CodeLen>::r3
            : ShiftLeft4<c4,c3,c2,c1,n>::r4,
        overflow = (n>=CodeLen && c4!=0)
    };
};
```

```
    || (n>=2*CodeLen && c3!=0)
    || (n>=3*CodeLen && c2!= 0)
    || (n>=4*CodeLen && c1!=0)
};
};
```

The overflow enumerator will be non-zero when some bits are lost off the left end of the leftmost Code of the multipart integer. This is not necessarily an error, so it is not flagged as such. Finally, let's wrap up the left shift facility with a clean interface.

```
template <Code c4, Code c3, Code c2, Code c1, Code n>
struct ShiftLeft {
    enum {
        code1 = ShiftLeftLong4<c4,c3,c2,c1,n>::r1,
        code2 = ShiftLeftLong4<c4,c3,c2,c1,n>::r2,
        code3 = ShiftLeftLong4<c4,c3,c2,c1,n>::r3,
        code4 = ShiftLeftLong4<c4,c3,c2,c1,n>::r4
    };
};
```

The implementation of multipart integer right shift is similar. Other bitwise operations can be implemented in a similar manner, but we need only left and right shift to implement `typeof`.

Generating a Gödel Number

The first step in mapping an expression's type to an integer involves using function template argument deduction to uncover the type of the expression.

```
template <typename T>
struct GN<T> genGN( T & );
```

The `genGN` template does not have to be defined, as this template function is “invoked” at compile time without being called. When we invoke `genGN`, the compiler performs argument deduction for us, and instantiates the `GN` (Gödel number) template with the type of the argument. As we noted in part one, the formal argument is declared to be a reference so that the `const` and `volatile` qualifiers will not be stripped from `T`.

The general (unspecialized) definition of the `GN` template is straightforward.

```
template <typename T>
struct GN {
    enum {
        code4 = 0,
        code3 = 0,
        code2 = 0,
```

```
    code1 = BaseType<T>::code,  
    len = BaseType<T>::len  
};  
GODEL_NUMBER_LENGTH_CHECK(len);  
};
```

This unspecialized template assumes that the type name `T` represents a base type, and so instantiates the `BaseType` template to obtain the unique integer value for that base type. The check on the length of the result is used to detect cases where the generated Gödel number overflows the multipart integer that holds it.

```
#define GODEL_NUMBER_LENGTH_CHECK(n) \  
    char xxx_[4*CodeLen-(n)+1]
```

If the generated number does overflow, then we will get a compile time error for attempting to declare an array of zero or negative bound and—one hopes—a compiler error message that mentions the macro name. The `BaseType` template generates a unique integer for the type using the registration mechanism described in the previous installment of this column.⁴

```
template <typename T>  
struct BaseType {  
    typedef DeQual<T>::R S;  
    enum {  
        base = TypeCode<S>::value,  
        code = base  
            | (IsConst<T>::result << ConstBasePos)  
            | (IsVol<T>::result << VolBasePos),  
        len = QualBaseLen // fixed length of a qualified base  
    };  
};
```

`BaseType` first strips off any type qualifiers, generates the unique code for the unqualified base type, then sets bits to represent the `const` and `volatile` qualifiers as appropriate. The stripping or detection of type qualifiers may be handled in an *ad hoc* manner, or by a more general facility. In this case, we were *ad hoc*.

```
template <typename T>  
struct IsConst {  
    enum { result = false }; };  
template <typename T>  
struct IsConst<const T> {  
    enum { result = true }; };  
  
template <typename T>
```

```
struct DeConst {
    typedef T R; };
template <typename T>
struct DeConst<const T> {
    typedef T R; };

// volatile similar to const...

template <typename T>
struct IsQual {
    enum { result = IsConst<T>::result
            || IsVol<T>::result }; };

template <typename T>
struct DeQual {
    typedef DeVol<DeConst<T>::R>::R R; };
```

Types with modifiers are handled by a number of different partial specializations of GN. As described in part one, we construct the code for a modified type by shifting the code for the unmodified type left, and appending the code corresponding to the modifier. Let's look at the partial specialization of GN for the pointer modifier.

```
template <typename T>
struct GN<T *> {
    enum {
        // generate base code...
        c1 = GN<T>::code1,
        c2 = GN<T>::code2,
        c3 = GN<T>::code3,
        c4 = GN<T>::code4,
        // ...shift it left...
        code4 = ShiftLeft<c4,c3,c2,c1,ModLen>::code4,
        code3 = ShiftLeft<c4,c3,c2,c1,ModLen>::code3,
        code2 = ShiftLeft<c4,c3,c2,c1,ModLen>::code2,
        // ...and append code for pointer modifier.
        code1 = ShiftLeft<c4,c3,c2,c1,ModLen>::code1 | Ptr,
        len = GN<T>::len+ModLen
    };
};
```

```

    GODEL_NUMBER_LENGTH_CHECK(len);
};

```

First, we generate the multipart integer that represents the pointed-to type. Then, we shift the multipart integer left by `ModLen`, the number of bits that a modifier occupies. The code for a pointer modifier, `Ptr`, is or-ed onto the right end of the multipart integer. Again, we keep track of the length of the generated multipart integer in order to detect overflow. Qualified pointer modifiers are handled in a similar way, but we also set bits to indicate `const`, `volatile`, or both.

```

template <typename T>
struct GN<T * const> {
    enum {
        // same as above...
        code1 = ShiftLeft<c4,c3,c2,c1,ModLen>::code1
            | Ptr+Const,
        len = GN<T>::len+ModLen
    };
    GODEL_NUMBER_LENGTH_CHECK(len);
};

```

Here, the choice to use addition instead of bitwise or to “add” the `Const` qualifier code to the `Ptr` modifier code is purely a stylistic one, since `code1` has been shifted left, and its rightmost `ModLen` bits are, therefore, all zeros. A bitwise or would have functioned just as well.

As we mentioned above, we don’t have space to show the entire implementation, but we can look at one of `GN`’s more complex specializations. Listing 1 shows the specialization of `GN` for unary functions. Recall that the encoding of a unary function follows the structure below.

Ret Type	Arg Type	Len	1	Fun Mod
----------	----------	-----	---	---------

The specialization of `GN` in Listing 1 first extracts the argument and return types of the function. It also extracts the type of the function as a whole, even though this typename is not used in the implementation of the template. It’s often a good idea when designing a template to provide more information than is strictly required for its implementation, since this information can be valuable both for debugging the template and for possible later use of the template in a manner that was not envisaged when it was first designed. This is the reason we earlier provided the ability to detect overflow in the left shift operation (through the presence of the `ShiftLeftLong4::overflow` enumerator), even though that feature is not used in this application.

We then recursively instantiate `GN` to generate the encodings for the argument and return types, and left-shift these encodings to the positions they will occupy in the final encoding of the function type as a whole. Finally, we use bitwise or to paste the return and argument types together, and append the length of the argument, followed by the number of arguments (one, since this is a unary function), and the code for a function modifier. As always, we calculate the total number of bits used by the encoding, and check

that we have not exceeded the maximum.

As an example, the encoding for the type `bool (*const*)(bool S::*)` can be determined by invoking `genGN`.

```
bool (*const*fp)(bool S::*);
const int c4 = genGN(fp).code4;
const int c3 = genGN(fp).code3;
const int c2 = genGN(fp).code2;
const int c1 = genGN(fp).code1;
```

If the class type `S` has been registered with the (decimal) value 31, the result is `[0X0040, 0X5F04, 0X5905, 0X0901]`, given the current, in some cases arbitrary, integer sizes and base type and modifier values used in the `GN` source code.

Observations and Confessions

In some ways, coding with template metaprogramming is liberating. Execution is performed at compile time, so there is no cost for code that would otherwise cry out for factoring if it were to be executed at runtime. In the partial specialization of `GN` for pointer modifiers, for example, we instantiated the `GN` and `ShiftLeft` templates multiple times with the same arguments. In the implementation of `ShiftLeft`, for clarity we layered the implementation based on both the length of the shift and on the number of constituent parts of the multipart integer being shifted. Clarity and ease of maintenance are always a consideration in any style of coding. In coding with templates these considerations are paramount, due to the difficulty of debugging generic code and the low cost of such an organization.

A confession is also in order. Like much bleeding-edge template metaprogramming, the implementation of `GN` has crossed over what can only be called the “Alexandrescu horizon,”⁵ and the compiler is unable to cope with some of its complexities. As a result, some aspects of the Gödel number generation do not work perfectly as of this writing. It is possible, of course, that some of the fault is with the implementation, but my preference is to blame the compiler instead.

In the next and final installment of this series, we’ll use template metaprogramming to implement compile time switch and if statements. We’ll then use these control structures to decode the Gödel numbers to extract the types that they encode, thereby completing our round-trip and implementing a portable `typeof` operator.

end of article

Listing 1

```
template <typename R, typename A>
struct GN<R (A)> {
    typedef R RetType;
    typedef A ArgType;
    typedef R FuncType( A );
    enum {
```

```
// calculate how much to shift ret and arg
argshift = ArgLenLen+ArgcountLen+ModLen,
arglen = GN<ArgType>::len,
retshift = arglen+argshift,

// get code for return type
r1 = GN<RetType>::code1,
r2 = GN<RetType>::code2,
r3 = GN<RetType>::code3,
r4 = GN<RetType>::code4,
retlen = GN<RetType>::len,

// get code for arg type
a1 = GN<ArgType>::code1,
a2 = GN<ArgType>::code2,
a3 = GN<ArgType>::code3,
a4 = GN<ArgType>::code4,

// shift arg type
sa4 = ShiftLeft<a4,a3,a2,a1,argshift>::code4,
sa3 = ShiftLeft<a4,a3,a2,a1,argshift>::code3,
sa2 = ShiftLeft<a4,a3,a2,a1,argshift>::code2,
sa1 = ShiftLeft<a4,a3,a2,a1,argshift>::code1,

// shift return type
sr4 = ShiftLeft<r4,r3,r2,r1,retshift>::code4,
sr3 = ShiftLeft<r4,r3,r2,r1,retshift>::code3,
sr2 = ShiftLeft<r4,r3,r2,r1,retshift>::code2,
sr1 = ShiftLeft<r4,r3,r2,r1,retshift>::code1,

// paste everything together
code4 = sr4 | sa4,
code3 = sr3 | sa3,
code2 = sr2 | sa2,
// note assumption below that ArgcountLen+Modlen
// < number of bits in Code!
```

```
code1 = srl | sal | (arglen<<(ArgcountLen+ModLen))
        | (1<<ModLen) | Fun,

// calculate total length
len = retlen+arglen+ArglenLen+ArgcountLen+ModLen
};
// check for overflow
GODEL_NUMBER_LENGTH_CHECK(len);
};
```

¹ An ancillary goal is to demonstrate to the College Board the continued viability of the C++ bitwise operators. But enough with the College Board already.

² For the interested reader, the source code for the portion of the `typeof` facility described in this article is available at <http://www.semantics.org/code.html>. The full implementation will be made available after the third (and final) part of this column.

³ The rules for the precision of enumerators are non-trivial. See the C++ standard, section 7.2.

⁴ S.C. Dewhurst, “A Bit-Wise Typeof, Part 1”, CUJ, August 2002.

⁵ After Andrei Alexandrescu, who had the cheek to publish a book of groundbreaking C++ programming techniques in advance of the existence of a compiler that could actually translate them. My reference to “the compiler” is generic, and may be more precisely stated as “the set of C++ compilers that I used to attempt to compile this code.”