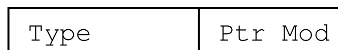
A Bit-Wise Typeof Operator, Part 3

by Stephen C. Dewhurst

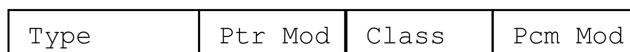
This is the third and final installment of a description of a mechanism for implementing a portable `typeof` operator. In Part 1, we described an approach for implementing `typeof` with a "round trip" mapping from a type to a structured integer encoding of the type and back again at compile time. In Part 2, we showed one way to implement compile time bitwise operations on integers of unbounded length, and showed how these multipart bitwise operations could be used to perform the encoding from type to integer. In this final part of the "`typeof`" series, we'll decode the integer to restore the original type. The round trip from type to integer to type will provide us with a portable `typeof` facility.[1]

Peeling Integers

In the previous installment, we converted a type into an integer through the use of partial specialization of the `GN` (Gödel number) template. An integer representing the type was built up by recursively shifting the integer left and appending additional modifiers. For example, the representation of a simple pointer type (say, `Type *`) would consist of the code for the base type shifted left and appended with the code for a pointer modifier:



The representation of a pointer to a member of that pointer type (say, `Type * Class::*`) would shift the previous representation left the appropriate number of bits, and append the code for the class type followed by the code for the pointer to member modifier:



Our job is now to recursively peel away these structured layers of the integer in order to recreate the original type. In the case of a pointer type, as in the first example above, we'll shift the integer right in order to peel off the pointer modifier code then recursively decode the remainder of the integer to produce the dereferenced type. Finally, we'll append a pointer modifier to restore the original type. In the case of our second example, we'll peel off both the pointer-to-member modifier code and the code for the class type, before recursively decoding the remainder of the integer. Finally, we'll append the pointer-to-member modifier to the resulting type.

Last time we decided to limit ourselves to four-part integers, though we would probably increase this in a production `typeof` implementation. The `GenType` template is, therefore, instantiated with four integral

codes that constitute the encoding of the type.

```
template <Code,Code,Code,Code>
struct GenType;
```

Compile Time Control Structures

The `GenType` template performs a compile time if-statement, instantiating either the `GenBaseType` or `GenModType` templates depending on whether the multipart integer used to instantiate it encodes a simple type or a modified type:

```
template <Code code4, Code code3, Code code2, Code code1>
struct GenType {
    typedef typename GenTypeImpl<ISBASECODE(code1),
            code4, code3, code2, code1>::Result GTR;
};
```

The implementations of the branches of the compile time if-statement are implemented as the primary and partially specialized versions of `GenTypeImpl`.

```
template <bool cond, Code code4, Code code3, Code code2, Code
code1>
struct GenTypeImpl {
    typedef typename
            GenBaseType<code1>::R Result;
};


template <Code code4, Code code3, Code code2, Code code1>
struct GenTypeImpl<false,code4,code3,code2,code1> {
    typedef typename
            GenModType<code4,code3,code2,code1>::MR Result;
};
```

Note that it was necessary to break out the implementation of `GenType` into a separate class. An alternative implementation might attempt to use a mechanism like Loki's `Select`[2] to perform the compile time conditional:

```
template <Code code4, Code code3, Code code2, Code code1>
struct GenType {
    typedef typename Select<
            ISBASECODE(code1),
            typename GenBaseType<code1>::R,
            typename GenModType<code4,code3,code2,code1>::MR
    >::Result GTR;
```

```
};
```

Unfortunately, this has the effect of instantiating both branches of the compile time if-statement, no matter what the value of the `ISBASECODE(code1)` conditional. Often this will result in an error if one of the branches is not legal C++ for a particular set of template arguments. Recall that all computation is being performed at compile time and (essentially) in parallel. Even though only one branch will be used as the result of the `Select`, both will be compiled. Moving each branch to a separate template ensures that only one will be.

Implementation of `GenBaseType` is trivial:

```
template <Code code>
struct GenBaseType {
    enum {
    base = code & BaseMask,
    isconst = code & ConstBaseQual,
    isvol = code & VolBaseQual
    };
    // decode base type
    typedef typename DeCode<base>::R B;
    // qualify it, if necessary
    typedef typename
        Select<isconst, const B, B>::Result CB;
    typedef typename
        Select<isvol, volatile CB, CB>::Result VCB;
    typedef VCB R;
};
```

The base code is extracted and stripped of any cv-qualifiers (`const` and/or `volatile`), then decoded using the registration mechanism described in the first part of this series.[3] Any cv-qualifiers are then reapplied.

Switches and Nested Switches

Type modifiers are handled by peeling off the code for the modifier and executing a compile time switch-statement on the code:

```
template <Code code4, Code code3, Code code2, Code code1>
struct GenModType {
    enum {
    mod = code1 & ModMask, // get modifier
    // shift modifier off
    c1 = ShiftRight<code4,code3,code2,code1,ModLen>::code1,
```

```
        c2 = ShiftRight<code4,code3,code2,code1,ModLen>::code2,
        c3 = ShiftRight<code4,code3,code2,code1,ModLen>::code3,
        c4 = ShiftRight<code4,code3,code2,code1,ModLen>::code4
        };
        // switch on modifier code
        typedef typename ModSwitch<mod,c4,c3,c2,c1>::MSR MR;
    };
```

The switch-statement is implemented as a set of partial specializations of the `ModSwitch` template. There is no need for an implementation of the primary `ModSwitch` template, though it must be declared before we can define partial specializations of it.

```
    template <Code mod,
       Code code4, Code code3, Code code2, Code code1>
    struct ModSwitch;
```

The implementations for most modifiers are straightforward. The case for a pointer modifier code is handled by recursively decoding the remainder of the type then appending the pointer modifier to the resulting type:

```
    template <Code code4, Code code3, Code code2, Code code1>
    struct ModSwitch<Ptr,code4,code3,code2,code1> { // *
        typedef typename
            GenType<code4,code3,code2,code1>::GTR *MSR;
    };
```

Pointers to class members are somewhat more complex:

```
    template <Code code4, Code code3, Code code2, Code code1>
    struct ModSwitch<Pcm,code4,code3,code2,code1> { // C::*
        enum {
        // get registered code for class
        classtype = code1 & BaseMask,
        // shift off the class code
        c1 = ShiftRight<code4,code3,code2,code1,BaseLen>::code1,
        c2 = ShiftRight<code4,code3,code2,code1,BaseLen>::code2,
        c3 = ShiftRight<code4,code3,code2,code1,BaseLen>::code3,
        c4 = ShiftRight<code4,code3,code2,code1,BaseLen>::code4
        };
        // get type of data member we're pointing to
        typedef typename GenType<c4,c3,c2,c1>::GTR MemType;
        // get class type
        typedef typename DeCode<classtype>::R ClassType;
```

```
    // append pointer to member modifier to member type
    typedef MemType ClassType::*MSR;
};
```

After extracting the registered type code for the class to which the pointer to member refers, the multipart integer is shifted right to remove the type code and leave the encoding for the type to which the pointer to member refers. This is recursively decoded (producing `MemType` in the code above), the class type is recovered from the registered type code extracted previously (`ClassType`), and the result is constructed by "pasting together" `MemType` and `ClassType` with a `typedef`.

Recall that the function modifier may indicate a function of zero, one, or more arguments. The encoding records the number of arguments, as well as the length, in bits, of each argument type. Encodings for functions taking zero, one, and two arguments are shown below:

Ret Type	0	Fun Mod

Ret Type	Arg Type	Len	1	Fun Mod

Ret Type	Arg2 Type	Arg1 Type	Len2	Len1	2	Fun Mod

Listing 1 shows the implementation for the function modifier. The implementation of `ModSwitch` for the function modifier code extracts the argument count and performs a nested switch-statement to the appropriate implementation of `FunSwitch`. Implementations for zero and one argument are shown.

Typeof

Finally, we must connect the part of our implementation that generates a Gödel number from a type to the part that generates a type from a Gödel number. Last time, we used the following functions to generate the encoding from the type:

```
template <typename T>
char (*genGN1( T & ))[GN<T>::code1+1]; // no definition
template <typename T>
char (*genGN2( T & ))[GN<T>::code2+1];
template <typename T>
char (*genGN3( T & ))[GN<T>::code3+1];
template <typename T>
char (*genGN4( T & ))[GN<T>::code4+1];
```

Note that we have incremented the value of each code by one in order to avoid the possibility of an illegal zero array index. This incremented value is later decremented to restore the correct value.

All we have to do is instantiate `GenType` with these integral values. Although they are generally to be avoided, a preprocessor macro is probably the best way to implement this:

```
#define NT_TYPEOF( e ) GenType< \
    sizeof(*genGN4(e))-1, \
    sizeof(*genGN3(e))-1, \
    sizeof(*genGN2(e))-1, \
    sizeof(*genGN1(e))-1  \
    >::GTR
#define TYPEOF( e ) typename NT_TYPEOF(e)
```

Unfortunately, the `typename` keyword may be used only within a template, as so we are forced to provide two versions of `typeof`. In most cases, the facility will be used within templates (since that's the only place we really have to ask what an expression's type is), so the more mnemonic name `TYPEOF` is used there. The name `NT_TYPEOF` is for use in non-template contexts.

Now that we have a fairly functional, portable `typeof` facility we can solve the original problem that motivated this `typeof` implementation; we can extract the type of an expression of (otherwise) unknown type. In the code below, we know the type of the container `cont` (it's `Cont`), but without `typeof` we wouldn't be able to uncover the type of its elements.

```
template <typename Cont>
void process( Cont &cont ) {
    TYPEOF(cont[0]) value( cont[0] ); // works!
    const int elemSize1 = sizeof( TYPEOF(cont[0]) ); // error!
    typedef TYPEOF( cont[0] ) Elem; // workaround…
    const int elemSize2 = sizeof( Elem ); // OK
    const int elemSize3 = sizeof( cont[0] ); // OK
   //…
```

The primary drawback of this implementation of `typeof` is the need to register user-defined types explicitly. For instance, if the element type of the container type `Cont` in the example above is (or is constructed with) an unregistered user-defined type, the use of `TYPEOF` will fail. Additionally, it may sometimes be necessary to work around syntactic difficulties incurred by a facility that is not a built-in operator, as shown above by the interaction of `TYPEOF` and `sizeof`. It would still be very useful to have a standard, built-in `typeof` operator in the C++ language.

However, our implementation of `typeof` illustrates some important techniques in template metaprogramming, and also illustrates the occasional utility of moving a difficult problem to a different solution domain. The `typeof` implementation moves between the domains of type manipulation and integer manipulation. In a future article, we'll examine a mechanism for implementing arbitrary precision compile time arithmetic by convincing the compiler to perform arithmetic operations on types.

Nolo Contendere

Alert reader and critic-at-large Terje Slettebø has identified an error in Part 1 of this series of columns, pointing out that the simple code used to demonstrate the "round trip" approach to the `typeof` implementation was illegal:

```
const int DoubleVal = 12; // code for double
//…
Int2Int<DoubleVal> typeCode(double &);
//…
template <int typecode> struct DeCode;
template <> struct DeCode<DoubleVal>
   { typedef double R;};
//…
DeCode<typeCode(cont[0]).value>::R // illegal!
         value( cont[0] );
```

The argument used to instantiate the `DeCode` template must be an integer constant-expression, and a constant-expression may not contain, directly, a function call. My compilers (and, admittedly, I) had a more liberal and non-standard view of the matter, however, and allowed the expansion because it was possible in this case to determine the value of the expression at compile time. In situations like this, we usually reach for `sizeof`:

```
char (*typeCode(double &))[DoubleVal];
//…
typename DeCode<sizeof(*typeCode(cont[0]))>::R // OK
         value( cont[0] );
```

In this modified version, the `typeCode` functions return a pointer to an array of characters, where the array bound is a compile time constant equal to the code of interest. It is legal to employ `sizeof` in an integer constant-expression, even if the `sizeof` expression contains a function call. Therefore we can extract the code as the `sizeof` the dereferenced return type of the function. This was the approach we took in the implementation of the `genGN` functions in Part 2 of this series.

Listing 1

```
template <Code argcount, Code code4, Code code3, Code code2,
Code code1>
struct FunSwitch; // no definition of primary

template <Code code4, Code code3, Code code2, Code code1>
struct FunSwitch<0,code4,code3,code2,code1> { // R ()
   typedef typename GenType<code4,code3,code2,code1>::GTR
RetType;
   typedef RetType FSR(void);
};

template <Code code4, Code code3, Code code2, Code code1>
```

```
struct FunSwitch<1,code4,code3,code2,code1> { // R (A)
    enum {
          arglen = code1 & ((1<<ArglenLen)-1),
          argshiftlen = (CodeLen*NumCodes)-(arglen+ArglenLen),

          rc1 =
ShiftRight<code4,code3,code2,code1,arglen+ArglenLen>::code1,
          rc2 =
ShiftRight<code4,code3,code2,code1,arglen+ArglenLen>::code2,
          rc3 =
ShiftRight<code4,code3,code2,code1,arglen+ArglenLen>::code3,
          rc4 =
ShiftRight<code4,code3,code2,code1,arglen+ArglenLen>::code4,

          t1 =
ShiftLeft<code4,code3,code2,code1,argshiftlen>::code1,
          t2 =
ShiftLeft<code4,code3,code2,code1,argshiftlen>::code2,
          t3 =
ShiftLeft<code4,code3,code2,code1,argshiftlen>::code3,
          t4 =
ShiftLeft<code4,code3,code2,code1,argshiftlen>::code4,

          ac1 =
ShiftRight<t4,t3,t2,t1,argshiftlen+ArglenLen>::code1,
          ac2 =
ShiftRight<t4,t3,t2,t1,argshiftlen+ArglenLen>::code2,
          ac3 =
ShiftRight<t4,t3,t2,t1,argshiftlen+ArglenLen>::code3,
          ac4 =
ShiftRight<t4,t3,t2,t1,argshiftlen+ArglenLen>::code4
    };
    typedef typename GenType<rc4,rc3,rc2,rc1>::GTR RetType;
    typedef typename GenType<ac4,ac3,ac2,ac1>::GTR ArgType;
    typedef RetType FSR( ArgType );
};

template <Code code4, Code code3, Code code2, Code code1>
struct ModSwitch<Fun,code4,code3,code2,code1> { // ()
    enum {
```

```
            argcount = code1 & ((1<<ArgcountLen)-1),
            c1 =
ShiftRight<code4,code3,code2,code1,ArgcountLen>::code1,
            c2 =
ShiftRight<code4,code3,code2,code1,ArgcountLen>::code2,
            c3 =
ShiftRight<code4,code3,code2,code1,ArgcountLen>::code3,
            c4 =
ShiftRight<code4,code3,code2,code1,ArgcountLen>::code4
    };
    typedef typename FunSwitch<argcount,c4,c3,c2,c1>::FSR MSR;
};
```

---

[1] We will, unfortunately, not have space to include the entire implementation of the `typeof` facility. The interested reader may find source code for the complete implementation at http://www.semantics.org/code.html. In particular, many details of the handling of the `void` type and implementation of pointers to cv-qualified member functions have had to be omitted here.

[2] Andrei Alexandrescu, *Modern C++ Design*, Addison Wesley 2001, p. 33.

[3] S. C. Dewhurst, "A Bit-Wise Typeof, Part 1," C/C++ Users Journal, 20(8), August 2002.