

# A Bit-Wise Typeof Operator, Part 1

*by Stephen C. Dewhurst*

Readers who braved the previous installment of this column may recall that I was greatly put out by the College Board's dismissal of bitwise operators as "not essential."<sup>1</sup> Usually I can work through emotional issues like this in a month or so, but due to a rather inconvenient change to this journal's editorial calendar, I'm writing this installment only two weeks after I submitted that one. I'm still steaming, and that means we're in for more bit manipulation.

As with the previous column, we'll be using the lower level (but essential!) bit manipulation operations in combination with more recent techniques. Last time, we used exclusive-or to create two-way pointers for an unusual STL container. This time, we'll use shifting and masking of raw bits combined with template metaprogramming to approximate a Gödel numbering for the C++ type system, which will then be used to implement a "bit-wise" `typeof` operator.<sup>2</sup>

## Type Algebras and Typeof

Most well designed compilers have a notion of a type algebra that is used to construct and manipulate types at compile time. The operations defined in the algebra are operators like `dereference`, `makereference`, `makearray`, etc. These operators are used to construct a name's type as its declaration is parsed, and to construct new types from existing types as names are manipulated in expressions. Note that these types are statically determined, compile time properties of names and expressions, so we can emulate the compiler's manipulation of types (to a large extent) with simple template metaprogramming techniques. In fact, it's pretty easy to manipulate a type at compile time once you've got one. The trouble is, it's not always easy to determine the type of an expression so that you'll have a type to manipulate.

```
template <typename Cont>
void process( const Cont &cont ) {
    Sometype value( cont[0] );
    //...
```

The trouble here is that, while we know that the type of `cont` is `Cont`, we have no notion what the type of `cont[0]` might be. The STL uses convention to solve the problem.

```
typename Cont::value_type value( cont[0] );
```

All STL-compliant containers can be queried for their element type. If we're not necessarily dealing with STL-compliant containers, and we're desperate, we can establish our own convention using some sort of ad hoc traits class.

```
template <class Cont>
```

```
struct ContainerTraits {
    typedef typename Const::value_type Etype;
    //...
};
template <class T>
struct ContainerTraits<MyContainer<T> > {
    typedef T Etype;
    //...
};
//...
typename ContainerTraits<Cont>::Etype value( cont[0] );
```

By default, our `ContainerTraits` assumes an STL-compliant container. We can augment the set of containers by providing a partial or complete specialization of the template. However, convention is costly to design and promulgate; that is, one not only has to establish a convention, one has to convince everyone else concerned to employ the convention. It would be much easier to dispense with convention and simply extract the type of the expression.

```
typeof(cont[0]) value( cont[0] );
```

Some C++ compilers do, in fact, provide a `typeof` operator as a nonstandard, nonportable extension. In this column and the next, we'll design and implement an effective, portable `typeof` facility, and make some points about the continued viability of bitwise operators along the way.

### Dysfunctional Deduction

One fairly standard approach to uncovering type information about expressions is function template argument deduction. Unfortunately, a template function returns a value of a particular type, not a type.

```
template <typename T>
struct Type {
    typedef T R; // what we want
    enum { // what we get
        isptr = IsPtr<T>::result,
        isconst = IsConst<T>::result,
        isDouble = IsSame<T,double>::result,
        //...
    };
};
template <typename T>
Type<T> extractType( T & )
    { return Type<T>(); }
```

This approach fails because there is no way to access the return type of the instantiated template function, only the return value.

```
extractType(cont[0])::R value( cont[0] ); // error!  
Select<extractType(cont[0]).isDouble, double, BadGuess>  
::Result value( cont[0] ); // not what we want
```

In effect, we can collect an arbitrary amount of information about an expression's type, but we can't access the actual type. That is, we can determine whether an expression's type is constant, a pointer, or a pointer to an array of 10. If we're willing to ask a direct question ("Are you a double?") we can, by chance, uncover the actual type. But there is no direct way to extract the type of an expression.

Frustrating.

Well, let's return a value, then.

```
template <typename T>  
struct TypeVal {  
    typedef T R;  
    static const long typecode;  
};  
template <typename T>  
const long TypeVal<T>::typecode = (long)&typecode;  
  
template <typename T>  
TypeVal<T> extractType( T )  
    { return TypeVal<T>(); }  
}
```

The value of `TypeVal<T>::typecode` is to be unique for each type `T` used to instantiate `TypeVal`, and can be used to distinguish types. (The value is unique because the static `typecode` contains its own address, and such addresses are unique.)

```
if( extractType( cont[0] ).typecode  
    == TypeVal<char **>::typecode )  
    //...
```

Unfortunately, the static `typecode` member is not an integer constant-expression, and any use of it, as in the conditional above, will be performed at runtime rather than compile time. This implies that this approach cannot be used to extract a (compile time) type. But mapping a type to an integer is not a bad idea.

### Mapping A Round Trip

If the root of our problem is obtaining a unique integer constant-expression for each type, we can explicitly associate each type with a unique integer, and use old-fashioned function overloading to map a type to its unique integral value.

```
const int DoubleVal = 12; // code for double
```

```
const int CharVal = 10; // code for char

template <int v>
struct Int2Int { enum { value = v }; };

Int2Int<DoubleVal> typeCode(double &);
Int2Int<CharVal> typeCode(char &);
```

Note that the overloaded `typeCode` functions cannot simply return an integral value, since the result of invoking the function would not be an integer constant expression. Instead, we return an object of a unique type that contains an enumerator with the type's code. In that way, the return value is known at compile time.

Once we've mapped a type to an integer constant-expression, we can perform the reverse mapping to regenerate the original type.

```
template <int typecode> struct DeCode;
template <> struct DeCode<DoubleVal>
    { typedef double R; };
template <> struct DeCode<CharVal>
    { typedef char R; };
```

The round trip from type to integer to type again allows us to extract the type of an expression.<sup>3</sup>

```
DeCode<typeCode(cont[0]).value>::R value( cont[0] );
```

If it so happens that the expression `cont[0]` has type `double` or `char`, we're in luck. Otherwise, we'll get a compile time error. Obviously, we should be able to recognize more types than just `double` and `char`. Unfortunately, a macro is the easiest way to accomplish this.

```
#define REGISTER( T, N )\
    Int2Int<N> typeCode(T &);\
    template <> struct DeCode<N> { typedef T R; }
```

This mechanism will allow us to determine the type of any expression whose type has been previously registered. The value of the codes associated with the types is immaterial. If the same code is associated with two different types, then a compile time error will occur, at least if the registrations occur in the same translation unit.<sup>4</sup>

```
REGISTER( float, 19 );
REGISTER( std::string, 72 );
REGISTER( char *, 39 );
REGISTER( std::deque<char *const *>, 9031 );
typedef int *(*FP)(std::string);
REGISTER( FP, 42 );
```

Note that the `typedef` in the last registration is required, since otherwise the macro expansion would result in erroneous syntax in the argument declaration of the `typeCode` function. Macros. What can

you do?<sup>5</sup>

Since we've already opened the floodgates, let's employ yet another macro to simplify the syntax of using our facility.<sup>6</sup>

```
#define TYPEOF( e ) DeCode<typeCode( e ).value>::R
```

The typeof facility is effective, in that it works (mostly), but it does require explicit registration of every type of interest.

```
template <typename Cont>
void process( const Cont &cont ) {
    TYPEOF(cont[0]) value( cont[0] );
    //...
```

One potential problem with this approach is that the `TYPEOF` an expression with reference type will not be a reference, but the dereferenced type. This actually mirrors the behavior of many of the `typeof` operator extensions mentioned earlier, and is conformant with the behavior of the standard `typeid` operator. (I don't like this aspect of `typeid` either.) However, recovering the fact that a name is actually a reference can be difficult or impossible, while it's trivial to strip a reference modifier from a type. As my old barber used to say, "I can take more off, but I can't put it back on!"

## Gödel Numbering

Explicit type registration can get pretty tedious, even if all the built in types are preregistered. For example, even if the type `int` is registered, we still have to register explicitly `int *`, `int **`, `const int`, `int (*) [10]`, `int (*) [11]`, and so on.

C++ compiler writers face a similar problem when generating external names for overloaded functions and similar complex language structures. The name of a function in C++ is really a composition of the function identifier and the formal argument types. Typically, the compiler will perform what is known as "name mangling" in order to produce a unique identifier for a (potentially overloaded) function. For example, a function declared as `void func( int, char *, double )` might be encoded as `func__FiPcd`. (Note that the return type does not participate in the encoding, although technically it could.) One of the nice things about name mangling is that the encoded name typically has a well-defined internal structure and is, therefore, invertible. That is, we can take the encoded name `func__FiPcd` and recognize that the function used to generate it took three formal arguments of types `int`, `char *`, and `double` in that order. It would be nice to be able to encode a general type in this way at compile time, then decode it to produce the type used to generate the encoded name. Unfortunately, effective compile time manipulation of character string literals is not possible.

However, effective compile time manipulation of integers is both possible and common. If we can find an effective way to "mangle" a type into an integer, then we should be able to decode the integer to restore the type.

This approach is reminiscent of the encoding process the logician Kurt Gödel described in the proof of his famous (and famously difficult) 1931 incompleteness theorem. As a preliminary step in the proof, Gödel describes a structured mapping from a sequence of symbols that represents a mathematical proof to a unique integer.<sup>7</sup> We'd like to construct an analogous mapping from C++ types to integers; that is, each type would map to a unique integral value. If the encoding process is deterministic and does not lose information, the internal logical structure of the integer can then be decoded to restore the type.

## Dewhurst — A Dysfunctional Typeof Operator

---

The encoding described here is just one of many reasonable possibilities. The representation of a base type like `int` or `std::string` will be a unique integer value set by the registration process described earlier. For convenience we will restrict the value so that it is representable in a specified number of bits. (A more sophisticated implementation might encode the length of the field in the field itself.)

A pointer modifier applied to a base type, as in `int *` or `std::string *` would be represented by shifting the base type encoding left an appropriate number of bits, and appending an integer code representing the pointer modifier.

Type	Ptr Mod
------	---------

Since the concatenation of base type and pointer modifier codes is itself a unique code for the pointer-modified base type, we can construct a pointer to a pointer (to a pointer, etc.) to the base type by repeating the process. The type qualifiers are handled for both base and modifier codes by setting specific bits within each field. For example, the integer code for the pointer modifier might be `0x01`, a constant pointer `0x09`, a volatile pointer `0x11`, and a constant volatile pointer `0x19`. The handling of the reference modifier is similar to that of the pointer modifier.

Other type modifiers are handled in a similar fashion. The array modifier must include the array bound.

Type	Bound	Arg Mod
------	-------	---------

A pointer to a class member must include the class type. The class type is simply the registered code for the class.

Type	Class	Pcm Mod
------	-------	---------

The function modifier is a bit more involved, and requires a little more finesse. Because the return type and argument types can be of any length, it's necessary to distinguish where one type ends and another begins. One way to do this is to record the number and length of the argument types. For example, function types with zero, one, and two arguments could be encoded as follows.

Ret Type	0	Fun Mod
----------	---	---------

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

Ret Type	Arg2 Type	Arg1 Type	Len2	Len1	2	Fun Mod
----------	-----------	-----------	------	------	---	---------

Note that each of these encodings is recursively defined. That is, subsidiary types may themselves be complex types.

Once we have constructed an integer from a type, it's trivial to decode the integer to restore the type. For example, if the lower N bits of the integer indicate a pointer modifier, then we recursively decode the remainder of the integer encoding (producing, say, type T1), and affix a pointer modifier to the result (producing T1 \*). If the lower N bits indicate a pointer to class member modifier, we decode the preceding class type (producing, say, the class type C), then recursively decode the remainder of the

integer encoding (producing, say, the type `T2`) and affix a pointer to member modifier to it (producing `T2 C::*`).

As a side point, note that the mapping between types and integers is not 1-1. A given type may be represented by different integers. For example, if we register `int *` directly with the `REGISTER` macro, both the registered value and a different value constructed from `int` and the pointer modifier will map to `int *`. This is not a problem unless the integer values are used directly. For example, the two different integer values for `int *` will not compare equal although they represent the same type. If we're careful to register only base types, however, each type will correspond to a single encoding. Of course, there will also be an infinite number of integer values that do not correspond to any type, because they will not be well formed according to the encoding rules.

That's basically it. The only major problem is the limited set of integers sizes available in C++. If the maximum integer size is 32 or 64 bits, this mechanism will not be able to encode complex types.

### Next Time

Next time, we'll get into serious bit manipulation. First, we'll solve the problem of performing logical operations on integers of unbounded length at compile time. Then we'll implement some rather involved recursive compile time bit-twiddling using template metaprogramming and send a big "so there!" to the College Board.

*end of article*

---

<sup>1</sup> S.C. Dewhurst, "Running Circles Round You, Logically," CUJ, June 2002.

<sup>2</sup> I originally started thinking about this problem after reading Koenig and Moo's interesting "Naming Unknown Types" (just perform a Koenig lookup in the February, 2002 CUJ), in which the authors mention briefly the problems posed by the unavailability of a portable `typeof` operator. (Beware that any article that is even indirectly concerned with Andy Koenig will inevitably contain a pun. A bad one, typically.)

<sup>3</sup> This is a simplified version of a technique described by Bill Gibbons in the online CUJ experts' forum, "A Portable 'typeof' Operator," November 2000, v.18, #11.

<sup>4</sup> This implies that it might be best to centralize registration. There's at least one deranged person on every project who would actually like to write a script to extract and centralize `REGISTRATIONS` from all the project source as a step in the build. Please leave me out of it.

<sup>5</sup> For one thing, you can wrap the `typedef` in a template and extract the syntactically simpler type name as a member of the template. Gibbons uses this approach in the article referenced above.

<sup>6</sup> Note that, contrary to standard advice, we have not parenthesized this `typeof` macro. What would happen if we did? See why I dislike macros?

<sup>7</sup> See Nagel and Newman's *Godel's Proof*, NYU Press, 1958 for a pleasant description of the numbering scheme.