EXPR_TYPE — An Implementation of typeof Using Current Standard C++

Anthony Williams

11th January 2002

Abstract

typeof is a much-sought-after facility that is lacking from current C++; it is the ability to declare a variable to have the same type as the result of a given expression, or make a function have the same type as an expression. The general idea is that typeof(some-expression) would be usable anywhere a type name could normally be used. This article describes a way of providing this ability within the realms of current C++.

Introduction

Imagine you're writing a template function that adds its parameters, such as the following:

```
template<typename T,typename U>
SomeResultType addValues(const T&t,const U&u)
{
    return t+u;
}
```

What type should the return value have? Obviously, the ideal choice would be "the type of t+u", but how can the compiler determine that type? The rules for the type of an additive expression involving only builtin types are not trivial — consider pointer arithmetic, integral type conversions, and integer conversion to floating-point type — so how *do* we do it? The Standard C++ Library only deals with the case where both function parameters have the same type, and forces the result to be the same type — e.g. std::plus, and std::min. Another possible solution is to hard code the rules for this particular operation, as Andrei Alexandrescu does for min and max [1].

The solution provided by some compilers as an extension, and widely considered to be a prime candidate for inclusion in the next version of the C++ standard is the typeof operator. The idea of typeof is that it takes an expression, and determines its type at compile time, without evaluating it, in much the same way as sizeof determines the size of the result of an expression without evaluating it. typeof could then be used anywhere a normal type name could be used, so we could declare addValues as:

```
template<typename T,typename U>
typeof(T()+U()) addValues(const T&t,const U&u);
```

The function parameters t and u aren't in scope at the point of declaration of addValues, so we use the default constructors for the types to generate values to add — the values used are unimportant, as the expression is never evaluated[1].

This is good — the return type of addValues is now correct for all combinations of types, but we now have the problem that we've used a non-standard extension, so the code is not portable. One compiler may use the name typeof, another __typeof__, and a third may not provide the extension at all. We therefore need a portable solution, that uses only Standard C++.

[1]There are better ways to generate suitable expressions, that don't rely on the type having a default constructor, but they will be dealt with later

Current Facilities

What facilities are there in Standard C++ that we can use? Firstly, we need a means of obtaining information about an expression without actually evaluating it. The only Standard facility for this is `sizeof`[2]. We therefore need a mechanism to ensure that expressions of different types yield different values for `sizeof`, so we can tell them apart, and we need a mechanism for converting a size into a type.

Another Standard C++ facility we can use to obtain information about a type is *function template argument type deduction.* By writing a function template which has a return type dependent on the argument type, we can encode information from the argument type into the return type any way we choose — given

```
template<typename T>
struct TypeToHelperInfo{};

template<typename T>
TypeToHelperInfo<T> typeOfHelper(const T& t);
```

we can then write the `TypeToHelperInfo` class template to provide any necessary information.

Bringing it Together

Converting a value into a type is easy — just create a class template with a value template parameter to accept the value which the type has been assigned. Then, specialize this template for each value/type combination, as in listing 1 — `Size1` and `Size2` are constants, and are the unique size values that relate to `Type1` and `Type2`, respectively, rather than `sizeof(Type1)` or `sizeof(Type2)`.

```
template<std::size_t size>
struct SizeToType
{};

template<>
struct SizeToType<Size1>
{
    typedef Type1 Type;
};

template<>
struct SizeToType<Size2>
{
    typedef Type2 Type;
};
```

Listing 1: Converting a size to a type

Converting a type into a size value is a bit more complex. If we have an expression `expr`, of type `T`, then `typeOfHelper(expr)` is of type `TypeToHelperInfo<T>`, if we use the signature of `typeOfHelper` from above. We can then specialize `TypeToHelperInfo`, so it has a distinct size for each distinct type. Unfortunately, it is not that simple — the compiler is free to add padding to `struct`s to ensure they get aligned properly, so we cannot portably control the size of a `struct`. The only construct in C++ which has a precisely-defined size is an array, the size of which is the number of elements multiplied by the size of each element. Given that `sizeof(char)==1`, the size of an array of `char` is equal to the number of elements in that array, which is precisely what we need. We can now specialize `TypeToHelperInfo` for each type, to contain an appropriately-sized array of `char`, as in listing 2.

[2]For non-polymorphic types, `typeid` also identifies the type of its operand at compile-time, without evaluating it, but the return type of `typeid` is very little use, as it is not guaranteed to have the same value for the same type, just an *equivalent* value.

```
template<>
struct TypeToHelperInfo<Type1>
{
    char array[Size1];
};

template<>
struct TypeToHelperInfo<Type2>
{
    char array[Size2];
};
```

Listing 2: Getting appropriately-sized arrays for each type

We can now simulate `typeof(expr)` with `SizeToType<sizeof(typeOfHelper(expr).array)>::Type`. In templates, we probably need to precede this with `typename`, in case `expr` depends on the template parameters. To ease the use, we can define an `EXPR_TYPE` macro that does this for us:

```
#define EXPR_TYPE(expr) SizeToType<sizeof(typeOfHelper(expr).array)>::Type
```

We also need to declare the appropriate specializations of `SizeToType` and `TypeToHelperInfo` for each type we wish to detect, so we define a `REGISTER_EXPR_TYPE` macro to assist with this, as in listing 3.

```
#define REGISTER_EXPR_TYPE(type,value)\
template<>\
struct TypeToHelperInfo<type>\
{\
    char array[value];\
};\
template<>\
struct SizeToType<value>\
{\
    typedef type Type;\
};
```

Listing 3: The `REGISTER_EXPR_TYPE` macro.

We can then declare the necessary specializations for all the basic types, and pointers to them, so our users don't have to do this themselves.

`const` Qualification

As it stands, with only the one `typeOfHelper` function template, `const` qualifications are lost. This may not be a problem, as `const`-qualification doesn't always have much significance with *value* types. However, this is a problem we can overcome[3] by providing two overloads of `typeOfHelper` instead of just the one:

```
template<typename T>
TypeToHelperInfo<T> typeOfHelper(T& t);
template<typename T>
TypeToHelperInfo<const T> typeOfHelper(const T& t);
```

We can then specialize the class templates for each distinct cv-qualified type — most easily done by modifying the `REGISTER_EXPR_TYPE` macro to register all four cv-qualified variants of each type with distinct values. Note

[3] for compilers that support partial ordering of function templates

that volatile qualification is automatically picked up correctly, since `T` will then be deduced to be "`volatile X`" for the appropriate type `X`. We only need these distinct overloads to allow the use of `EXPR_TYPE` with expressions that return a non-`const` temporary, since with only a single function taking a `T&`, `T` is deduced to be the non-`const` type, and temporaries cannot bind to non-`const` references. With both overloads, the temporary can be bound to the overload that takes `const T&`. The result is that temporaries are deduced to be `const`.[4]

In the final implementation, all the classes and functions are in namespace `ExprType`, to avoid polluting the global namespace, and the macro definitions have been adjusted accordingly.

Restrictions

The most obvious restriction is that this only works for expressions that have a type for which we have specialized `SizeToType` and `TypeToHelperInfo`. This has the consequence that we cannot define a specialization for `std::vector` in general; we have to define one specialization for `std::vector<int>`, and another for `std::vector<double>`. Also, in order to avoid violating the One Definition Rule, the user must ensure that the same value is used for the same type in all translation units that are linked to produce a single program. This includes any libraries used, so when writing library code that uses `EXPR_TYPE`, it is probably best to put the templates in a private namespace, to isolate them from the rest of the program, and avoid the problem.

Also, `EXPR_TYPE` cannot tell the difference between an *lvalue* and an *rvalue*, or between a reference and a value, except that *rvalues* are always `const`, whereas *lvalues* may not be — given

```
int f1();
int& f2();
const int& f3();
int i;
```

`EXPR_TYPE(f2())` and `EXPR_TYPE(i)` are `int`, whereas `EXPR_TYPE(f1())`, `EXPR_TYPE(f3())` and `EXPR_TYPE(25)` are `const int`. The reason for this is that the mechanism used for type deduction — function template argument type deduction — can only distinguish by cv-qualification, and *rvalues* are mapped to `const` references. This means you cannot use `EXPR_TYPE` to pass references — you must explicitly add the `&` where needed, though this can be done automatically for non-`const` references. It also means that you may have to strip the `const` if the expression results in an *rvalue*, and you wish to declare a non-`const` variable of that type, using something like `boost::remove_const` [3].

Finally, `EXPR_TYPE` cannot be used for expressions with `void` type, such as functions returning `void`. This is because, though references to incomplete types are permitted in general, references to `void` are explicitly not permitted.

Revisiting the Example

To come back to the example from the introduction, we can now implement our `addValues` template as:

```
template<typename T,typename U>
typename EXPR_TYPE(T()+U()) addValues(const T& t,const U& u)
{
    return t+u;
}
```

However, this still relies on the types `T` and `U` having default constructors. We can avoid this restriction by declaring a `makeT` function template:

[4]Some compilers that don't support partial ordering of function templates, also allow the binding of temporaries to non-`const` references, so we only need supply a single function, taking `T&`, in which case temporaries are deduced to be non-`const`.

```
template<typename T>
T makeT();
```

and then using this template in the parameter for `EXPR_TYPE` — `EXPR_TYPE(makeT<const T&>()+makeT<const U&>())`. This is a useful technique, whenever one is writing such an expression, where all we want is its type, or size — since the expressions using `makeT` are never evaluated, there is no need to provide a definition of `makeT`; consequently using `makeT` imposes no restrictions on the types.

Further Examples

This technique is useful in any scenario where you wish to declare an object of a type related to the type of something else, e.g. declaring a pointer to another variable

```
int i;
EXPR_TYPE(i) * j=&i; // j is "int *"
```

or supporting containers that may or may not declare `iterator` and `const_iterator` `typedef`s.

```
template<typename Container>
void func(const Container& container)
{
    for(typename EXPR_TYPE(container.begin()) it=container.begin();
        it!=container.end();++it)
    { // do something
    }
}
```

It can also be used for such things as implementing `min` and `max` function templates:

```
template<typename T,typename U>
struct MinHelper
{
    typedef typename EXPR_TYPE((makeT<const T&>()<makeT<const U&>())?makeT<const T&>():mak
};

template<typename T,typename U>
typename MinHelper<T,U>::Type min(const T& t,const U& u)
{
    return (t<u)?t:u;
}
```

In general, it is of most use in template code, with expressions where the type depends on the template parameters in a non-trivial fashion. This allows the writing of more generic templates, without complex support machinery.

It must be noted, however, that the user must ensure that all types to be deduced have been registered in advance. The library can facilitate this by declaring all the builtin types, and some compound types (e.g. `char*`, `const double*`, etc.), but class types and more obscure compound types (such as `char volatile**const*`) must be declared explicitly.

Summary

The macros and templates that make up the `EXPR_TYPE` library enable a `typeof`-like mechanism using Standard C++. The only cost is the maintenance burden placed on the user, to ensure there is a one-to-one correspondence between types and size-values across each project, and the only restriction is that it cannot tell the difference between lvalues and rvalues, and cannot detect `void`, though all cv-qualifications are identified.

This powerful mechanism is another tool in the Generic Programming toolbox, enabling people to write generic code more easily.

The code presented here will be available with the online version of the article at http://cplusplus.anthonyw.cjb.net/articles.html.

References

[1] Andrei Alexandrescu. Generic<Programming>: Min and Max Redivivus. *C/C++ Users Journal*, 19(4), April 2001. Available online at http://www.cuj.com/experts/1904/alexandr.htm.

[2] Bill Gibbons. A Portable "typeof" operator. *C/C++ Users Journal*, 18(11), November 2000.

[3] The Boost Team. Boost c++ libraries. See http://www.boost.org.