Existing C++ Compile-Time Reflection Demo Of Built-In Type Introspection Using Compile-Time Strings, Variable Templates and Type Deduction

Project: Programming Language C++ - SG7 Reflection
Reply-To: Andrew Tomazos <andrewtomazos@gmail.com>
Date: 21 May 2014
URL: http://www.tomazos.com/angloname.pdf

Table of Contents

- Abstract
- Goal
- Implementation
  - Compile-Time Strings
  - angloname<T> Variable Template
    - Fundamental Types
    - Cv-Qualified Types
    - Pointers and References
    - Arrays
    - Functions
    - Classes, Unions and Enumerations
- Conclusion

Abstract

We demonstrate how to use existing C++14 reflection features in order to solve the following toy problem:

Write a constexpr variable template `angloname<T>` that behaves like a string literal that holds the *anglicized name* of the type T.

The *anglicized name* of a type is how it is described in canonical prose in the standard. For example `angloname<int*>` should behave like the string literal `"pointer to int"`. The content of the string literal is calculated during translation (at compile-time) by using type deduction to recursively break apart the type and then uses a compile-time string helper class to assemble the name.

The purpose of this demo is to prime and motivate the reader for the type introspection extensions for user-defined types we will be proposing at Rapperswil. **We show here that we already have in C++ complete compile-time type introspection of the fundamental types,**

all built-in compound types (including every kind of function type and the function parameter list) and all cv-qualified versions thereof. What is missing is better type introspection of user-defined types (classes, unions, enumerations), currently these types are essentially opaque handles.

Goal

Once `angloname<T>` is implemented (see Implementation below) we will be able to perform tests like the following:

```
static_assert(angloname<char16_t> == "char16_t", "");

static_assert(angloname<signed long> == "long int", "");

static_assert(angloname<int*> == "pointer to int", "");

using T1 = long signed const int long* (* volatile &&) [13];
static_assert(angloname<T1> == "rvalue reference to volatile "
    "pointer to array of 13 pointer to const long long int", "");

using T2 = char16_t (char, float) volatile &&;
static_assert(angloname<T2> == "function of ( char, float ) "
             "volatile && returning char16_t", "");
```

The `static_assert` proves that the comparison is made at compile-time. `angloname<T>.value` can be used in all the same ways as a string literal. It is an array of char of static storage duration defined with constexpr holding null-terminated UTF-8 text. A pointer to it is an address constant expression. An lvalue-to-rvalue conversion is permitted on its elements within a core constant expression.

Implementation

The implementation is in two parts. First we will define a compile-time string class and some helper functions for it. Second we will define our `angloname<T>` variable template.

Compile-Time Strings

A built-in string literal is an array of char. Because we can't pass, initialize or return arrays by value we will wrap our array type in a class type called `strlit` (much like `std::array`):

```
template<size_t N>
struct strlit
{
```

```
      constexpr strlit() : value{0} {}
      char value[N];
};
```

Next we create a constexpr factory function `make_strlit` that can take a built-in string literal and return a strlit:

```
template<size_t N>
constexpr strlit<N>
make_strlit(const char (&S)[N])
{
      strlit<N> R;

      for (size_t i = 0; i < N; i++)
            R.value[i] = S[i];

      return R;
}
```

For arrays of known bound we will need to convert the size into a string. For example `angloname<float[42]>` is `"array of 42 float"`. To convert that 42 to a string first we write a function `count_digits` that determines the number of decimal digits a number has (to calculate how many chars we will need):

```
constexpr size_t
count_digits(size_t N)
{
      if (N == 0)
            return 1;

      size_t T = 0;

      while (N > 0)
      {
            N /= 10;
            T++;
      }

      return T;
}
```

Then we will use it to write a function template `to_strlit<n>()` that returns a string literal of its non-type template parameter `n`. For example, `to_strlit<42>()` returns `"42"`.

```
template<size_t n>
constexpr strlit<count_digits(n)+1>
to_strlit()
{
	strlit<count_digits(n)+1> r;

	size_t N = n;

	if (N == 0)
	{
		r.value[0] = '0';
		r.value[1] = '\n';
	}

	constexpr size_t D = count_digits(n);

	r.value[D] = '\0';

	for (size_t i = D-1; i != size_t(-1); i--)
	{
		r.value[i] = '0' + N % 10;
		N /= 10;
	}

	return r;
}
```

Next we will define a constexpr binary `operator+` on strlits to perform concatenation. We will use this to assemble the different parts of the `angloname` for compound types:

```
template<size_t N, size_t M>
constexpr strlit<N+M-1>
operator+(const strlit<N>& a, const strlit<M>& b)
{
	strlit<N+M-1> c;

	for (size_t i = 0; i < N-1; i++)
		c.value[i] = a.value[i];

	for (size_t i = 0; i < M; i++)
		c.value[N-1+i] = b.value[i];
```

```
      return c;
}
```

Finally we will define a constexpr equality function to compare strlits with string literals. This is used to test angloname (see Goal above):

```
template<size_t N, size_t M>
constexpr bool
operator==(const strlit<N>& a, const char (&S)[M])
{
      if (N != M)
            return false;

      for (size_t i = 0; i < N; i++)
            if (a.value[i] != S[i])
                  return false;

      return true;
}
```

angloname<T> Variable Template

We are now ready to define the `angloname<T>` variable template. First we define the primary template. It will never be the best match for any type so we assign it to the string literal "undefined". (Authors note: Richard Smith and I once discussed this issue on SO that there doesn't seem to be a good way to leave a primary variable template undefined as there is for class templates by leaving the primary incomplete.)

```
template<typename T, class Enable = void>
constexpr auto angloname = make_strlit("undefined");
```

Next we write a series of partial specializations and explicit specializations for each family of types in the C++ type system.

Fundamental Types

There are 20 fundamental types in the C++ type system. We simply write an explicit specialization of angoname for each:

```
template<> constexpr auto angloname<signed char>
                    = make_strlit("signed char");
```

```
template<> constexpr auto angloname<short int>
                    = make_strlit("short int");

template<> constexpr auto angloname<int>
                    = make_strlit("int");

template<> constexpr auto angloname<long int>
                    = make_strlit("long int");

template<> constexpr auto angloname<long long int>
                    = make_strlit("long long int");

template<> constexpr auto angloname<unsigned char>
                    = make_strlit("unsigned char");

template<> constexpr auto angloname<unsigned short int>
                    = make_strlit("unsigned short int");

template<> constexpr auto angloname<unsigned int>
                    = make_strlit("unsigned int");

template<> constexpr auto angloname<unsigned long int>
                    = make_strlit("unsigned long int");

template<> constexpr auto angloname<unsigned long long int>
                    = make_strlit("unsigned long long int");

template<> constexpr auto angloname<wchar_t>
                    = make_strlit("wchar_t");

template<> constexpr auto angloname<char>
                    = make_strlit("char");

template<> constexpr auto angloname<char16_t>
                    = make_strlit("char16_t");

template<> constexpr auto angloname<char32_t>
                    = make_strlit("char32_t");

template<> constexpr auto angloname<bool>
                    = make_strlit("bool");
```

```
template<> constexpr auto angloname<float>
                    = make_strlit("float");

template<> constexpr auto angloname<double>
                    = make_strlit("double");

template<> constexpr auto angloname<long double>
                    = make_strlit("long double");

template<> constexpr auto angloname<void>
                    = make_strlit("void");

template<> constexpr auto angloname<std::nullptr_t>
                    = make_strlit("std::nullptr_t");
```

Cv-Qualified Types

Next we write partial specializations for the three cv-qualified versions of a type: `const T`, `volatile T` and `const volatile T`:

```
template<typename T> constexpr auto angloname<const T>
 = make_strlit("const ") + angloname<T>;

template<typename T> constexpr auto angloname<volatile T>
 = make_strlit("volatile ") + angloname<T>;

template<typename T> constexpr auto angloname<const volatile T>
 = make_strlit("const volatile ") + angloname<T>;
```

Pointers and References

Then we write partial specializations for the pointer, pointer-to-member and reference types: `T*`, `T C::*`, `T&` and `T&&`:

```
template<typename T> constexpr auto angloname<T*>
 = make_strlit("pointer to ") + angloname<T>;

template<class C, typename T> constexpr auto angloname<T C::*>
 = make_strlit("pointer to member of class ") + angloname<C>
 + make_strlit(" of type ") + angloname<T>;
```

```
template<typename T> constexpr auto angloname<T&>
 = make_strlit("lvalue reference to ") + angloname<T>;

template<typename T> constexpr auto angloname<T&&>
 = make_strlit("rvalue reference to ") + angloname<T>;
```

Arrays

Next partial specializations for arrays of known bound `T[N]` and arrays of unknown bound `T[]`. Here we see where we use `to_strlit<N>()`:

```
template<typename T> constexpr auto angloname<T[]>
 = make_strlit("array of unknown bound of ") + angloname<T>;

template<typename T, size_t N> constexpr auto angloname<T[N]>
 = make_strlit("array of ") + to_strlit<N>()
 + make_strlit(" ") + angloname<T>;
```

Functions

Function types involve a little more work than the other types, but can be completely reflected with pure library type deduction as we will demonstrate. This includes enumerating the parameters.

Firstly, to hold the function parameter list we will need a substring that forms a comma-separated list. For example the type `int(char, float)` must be anglicized to `"function of ( char, float ) returning int"` - so to form the content between the parenthesis we will define a `angloname_list<T1, T2, ..., TN>` variable variadic template that simply concatenates the `angloname<Ti>` of each of its type parameters comma-delimited.

First the empty list:

```
template<typename... P> constexpr auto angloname_list
 = make_strlit("");
```

Then the list of one item:

```
template<typename T> constexpr auto angloname_list<T>
 = angloname<T>;
```

Then for lists of two or more items we defined them once recursively:

```
template<typename T, typename... P>
constexpr auto angloname_list<T, P...>
 = angloname<T> + make_strlit(", ") + angloname_list<P...>;
```

(Note this head-tail style is not necessarily the only or most efficient way (O(N) instantiations), we just use it here for simplicity.)

Now we are ready to define the first partial specialization for simple function types:

```
template<typename R, typename... P>
constexpr auto angloname<R(P...)>
 = make_strlit("function of ( ") + angloname_list<P...>
 + make_strlit(" ) returning ") + angloname<R>;
```

Additionally, function types also vary based on the trailing parts of the type that appertain to the cv-qualification (no cv qualification, const, volatile and const volatile) and ref-qualification (none, & or &&) of the implicit object parameter of member functions, and also as to whether or not they take a variable number of arguments (the C-style ellipsis). We specialize angloname<T> for each of the cross product of these variances (4 x 3 x 2 = 24 specializations):

```
template<typename R, typename... P>
constexpr auto angloname<R(P...) const>
 = make_strlit("function of ( ") + angloname_list<P...>
 + make_strlit(" ) const returning ") + angloname<R>;

template<typename R, typename... P>
constexpr auto angloname<R(P...) volatile>
 = make_strlit("function of ( ") + angloname_list<P...>
 + make_strlit(" ) volatile returning ") + angloname<R>;

template<typename R, typename... P>
constexpr auto angloname<R(P...) const volatile>
 = make_strlit("function of ( ") + angloname_list<P...>
 + make_strlit(" ) const volatile returning ") + angloname<R>;

template<typename R, typename... P>
constexpr auto angloname<R(P...) &>
 = make_strlit("function of ( ") + angloname_list<P...>
 + make_strlit(" ) & returning ") + angloname<R>;

template<typename R, typename... P>
```

```
constexpr auto angloname<R(P...) const &>
 = make_strlit("function of ( ") + angloname_list<P...>
 + make_strlit(" ) const & returning ") + angloname<R>;

template<typename R, typename... P>
constexpr auto angloname<R(P...) volatile &>
 = make_strlit("function of ( ") + angloname_list<P...>
 + make_strlit(" ) volatile & returning ") + angloname<R>;

template<typename R, typename... P>
constexpr auto angloname<R(P...) const volatile &>
 = make_strlit("function of ( ") + angloname_list<P...>
 + make_strlit(" ) const volatile & returning ") + angloname<R>;


template<typename R, typename... P>
constexpr auto angloname<R(P...) &&>
 = make_strlit("function of ( ") + angloname_list<P...>
 + make_strlit(" ) && returning ") + angloname<R>;

template<typename R, typename... P>
constexpr auto angloname<R(P...) const &&>
 = make_strlit("function of ( ") + angloname_list<P...>
 + make_strlit(" ) const && returning ") + angloname<R>;

template<typename R, typename... P>
constexpr auto angloname<R(P...) volatile &&>
 = make_strlit("function of ( ") + angloname_list<P...>
 + make_strlit(" ) volatile && returning ") + angloname<R>;

template<typename R, typename... P>
constexpr auto angloname<R(P...) const volatile &&>
 = make_strlit("function of ( ") + angloname_list<P...>
 + make_strlit(" ) const volatile && returning ") + angloname<R>;

template<typename R, typename... P>
constexpr auto angloname<R(P..., ...)>
 = make_strlit("function of ( ") + angloname_list<P...>
 + make_strlit(", ... ) returning ") + angloname<R>;

template<typename R, typename... P>
constexpr auto angloname<R(P..., ...) const>
 = make_strlit("function of ( ") + angloname_list<P...>
```

```
 + make_strlit(", ... ) const returning ") + angloname<R>;

template<typename R, typename... P>
constexpr auto angloname<R(P..., ...) volatile>
 = make_strlit("function of ( ") + angloname_list<P...>
 + make_strlit(", ... ) volatile returning ") + angloname<R>;

template<typename R, typename... P>
constexpr auto angloname<R(P..., ...) const volatile>
 = make_strlit("function of ( ") + angloname_list<P...>
 + make_strlit(", ... ) const volatile returning ") + angloname<R>;

template<typename R, typename... P>
constexpr auto angloname<R(P..., ...) &>
 = make_strlit("function of ( ") + angloname_list<P...>
 + make_strlit(", ... ) & returning ") + angloname<R>;

template<typename R, typename... P>
constexpr auto angloname<R(P..., ...) const &>
 = make_strlit("function of ( ") + angloname_list<P...>
 + make_strlit(", ... ) const & returning ") + angloname<R>;

template<typename R, typename... P>
constexpr auto angloname<R(P..., ...) volatile &>
 = make_strlit("function of ( ") + angloname_list<P...>
 + make_strlit(", ... ) volatile & returning ") + angloname<R>;

template<typename R, typename... P>
constexpr auto angloname<R(P..., ...) const volatile &>
 = make_strlit("function of ( ") + angloname_list<P...>
 + make_strlit(", ... ) const volatile & returning ")
 + angloname<R>;

template<typename R, typename... P>
constexpr auto angloname<R(P..., ...) &&>
 = make_strlit("function of ( ") + angloname_list<P...>
 + make_strlit(", ... ) && returning ") + angloname<R>;

template<typename R, typename... P>
constexpr auto angloname<R(P..., ...) const &&>
 = make_strlit("function of ( ") + angloname_list<P...>
 + make_strlit(", ... ) const && returning ") + angloname<R>;
```

```
template<typename R, typename... P>
constexpr auto angloname<R(P..., ...) volatile &&>
 = make_strlit("function of ( ") + angloname_list<P...>
 + make_strlit(", ... ) volatile && returning ") + angloname<R>;

template<typename R, typename... P>
constexpr auto angloname<R(P..., ...) const volatile &&>
 = make_strlit("function of ( ") + angloname_list<P...>
 + make_strlit(", ... ) const volatile && returning ")
 + angloname<R>;
```

Classes, Unions and Enumerations

Finally, we create a stub implementation for the user-defined types. To distinguish the three families we use std::enable_if_t and the three type traits std::is_class, std::is_union and std::is_enum. We then need to also assert that they are not cv-qualfied as the type traits will pick up cv-qualified versions and we already handle cv-qualfied types separately (see above Cv-Qualified Types). For each family we simply output “some class type”, “some union type” and “some enumeration type”. We see here the motivation for improvement of C++ type introspection. We would like to be able to determine the name of these types at compile-time, and also to be able to reflect into their various kinds of members and base classes.

```
template<typename E>
constexpr auto angloname<E, enable_if_t<is_enum<E>::value
                              && !is_const<E>::value
                              && !is_volatile<E>::value>>
 = make_strlit("some enumeration type");

template<typename C>
constexpr auto angloname<C, enable_if_t<is_class<C>::value
                              && !is_const<C>::value
                              && !is_volatile<C>::value>>
 = make_strlit("some class type");

template<typename U>
constexpr auto angloname<U, enable_if_t<is_union<U>::value
                              && !is_const<U>::value
                              && !is_volatile<U>::value>>
 = make_strlit("some union type");
```

Conclusion

The goal has now been achieved and angloname<T> is appropriately specialized for the entire C++ type system.

The code for this demo has been extracted into a single translation unit standalone C++14 program that is available at http://www.tomazos.com/angloname.cpp

To compile it you will need a C++14 compiler or clang trunk with -std=c++1y will do.

This PDF is available from http://www.tomazos.com/angloname.pdf