Dnnnn=yy-nnnn

Emil Madsen

Doc No: Dnnnn=yy-nnnn
Revision: 1.2
Date: 2013-12-04
Project: Programming Language C++, Library Evolution Working Group
Author: Emil Madsen
Reply to: Emil Madsen
Email: skeen@cs.au.dk

A proposal to add call traits to the Standard Library

Contents

1 Introduction

Compile-time reflection capabilities are one of the cornerstones of C++ template meta-programming. TR1 brought us the `<type_traits>` header, and thereby provided us with a way to query information about types, and their relationships.

This proposal aims to add support for compile-time retrieval of return-type, arity, parameter types, and such for various callable types; e.g. function pointers (free, static, and non-static), functors and lambdas. In general everything implementing the call operator.

That is a way to query information about callable types.

The proposed interface is inspired from the `<type_traits>` header, a listing of the proposed traits can be found in the technical specification.

A simple usage example can be found in the motivation section.

2 Motivation

Generic programming (decoupling of algorithm and type) has become the recommended way for providing reusable code.

However, there are times, in which we cannot provide a truly generic solution, and where we are forced to take type information into account, and hence we need a generic way to query information about the instantiated type(s).

This is where traits come into play, traits provide us with a way to have type specific information in a generic context. This is currently available for types using the `<type_traits>` header, however no similiar solution currently exists for callable types.

The motivation for this proposal is to 'correct' this, by provide what the `<type_traits>` header does for types, but for callable types.

Currently, when taking a callable type, by an unrestricted template, or when storing it using the auto keyword, one looses absolutely all information about the type, that is we're in an unknown state, about the callable type, and there's currently no standard way for querying any of this information, if one needs to.

Being able to query this kind of information, at compile-time will allow one to generate definitions, and to support generic code, with type specific behavior.

A pratictal toy example of this would be the following;

```
<code>
template <typename Callable, typename... Args>
struct converted_std_function_type_worker
{
    template <typename, typename>
    struct internal_struct;

    template <typename T, T... S>
    struct internal_struct<T, std::integer_sequence<T, S...>>
    {
        using std_function =
            std::function<typename std::return_type<Callable, Args...>(
                typename std::parameter_type<S, Callable, Args...>...)>;
    };
};

template <typename Callable, typename... Args>
using converted_std_function_type =
    typename converted_std_function_type_worker<Callable, Args...>::
        template internal_struct<size_t,
            std::make_integer_sequence<size_t,
                std::arity_integral<Callable, Args...>::value>>::
                    std_function;

template <typename Callable, typename... Args>
auto make_stdfunction(Callable c, Args... args)
    -> converted_std_function_type<Callable, Args...>
{ return c; }
</code>
```

Note: `std::integer_sequence<T, S...>` and friends refer to the concepts from N3493. The above code allows one to convert any callable type to a std::function, by extracting the required template information from the callable type at compile time.

Usage code may look like;

```
<code>
auto func = make_stdfunction(&function);
static_assert(std::is_same<decltype(func),
              std::function<int(int)>>::value, "");

auto lambda = make_stdfunction([](int){ return 'C'; });
static_assert(std::is_same<decltype(lambda),
              std::function<char(int)>>::value, "");
</code>
```

The above is of course just a simple toy example, for the purpose of this text. Plenty of other, possibly more useful examples can be thought up.

3 Scope

The people who'll likely be the users of this proposed extension to the standard library, are generally speaking library writers and other generic code developers.

Who are experienced C++ developers.

4 Impact On the Standard

This proposal can be implemented as a pure library extension, but it may be compiler supported as the implementation is left as a detail, and compiler and standard library vendors are free to implement the traits interface in any way they see fit.

The proposal does not require changes to any standard classes, functions or headers. It does however introduce a new header, namely `call_traits`.

5 Design Decisions

The interface design is chosen to match the `type_traits` library. As for naming, nothing is settled yet, as this is still a discussion paper.

6 Technical Specifications

6.1 Interface

I'm open to adding more traits to the interface, assuming they are implementable in a pure library solution.
Note, all implementations are denotated with `...`, as they are implementation specific. For a fill in, download the library referenced in 'Implementatation' sub-section.

6.1.1 Arity

Arity is the measure for the number of arguments a given function takes.

(1) Return the arity (e.g. number of arguments) for a given callable type.

(2) Return whether the callable type has the queried number of arguments.

```
// (1)
template <typename Callable, typename... Args>
struct arity
```

```
    : std::integral_constant<size_t, ...>
{
};

// (2)
template <typename Callable, typename... Args>
struct is_nullary
    : std::integral_constant<bool, (arity<Callable, Args...>::value == 0)>
{
};
template <typename Callable, typename... Args>
struct is_unary
    : std::integral_constant<bool, (arity<Callable, Args...>::value == 1)>
{
};
template <typename Callable, typename... Args>
struct is_binary
    : std::integral_constant<bool, (arity<Callable, Args...>::value == 2)>
{
};
template <typename Callable, typename... Args>
struct is_ternary
    : std::integral_constant<bool, (arity<Callable, Args...>::value == 3)>
{
};
```

Note; In the case of varying arguments (C-style ellipsis), `std::arity` returns the number of actual arguments.
Note; As default arguments are not a part of a function type (8.3.6), these will always be 'ignored', in terms of `std::arity`.
Note; For member functions, this will return the actual number of arguments, that is without the implicit object parameter, as this is handled seperately by the traits; `is_member_function`, and `implicit_object_type`.

6.1.2 Has Varying Arguments

Return whether or not the callable type, accepts a varying number of arguments (e.g. C style variadics, `void func(...)`).

```
template <typename Callable, typename... Args>
struct has_varying_arguments
    : std::integral_constant<bool, ...>
{
};
```

Note; Has nothing to do with variadic templates.

6.1.3 Is const function

Return whether the given member function, is declared const.

```
template <typename Callable, typename... Args>
struct is_const_function
    : std::integral_constant<bool, ...>
{
};
```

Note; Only ever non-false for non-static member functions.

6.1.4 Is function pointer

Return whether the callable type is a function pointer.

```
template<typename Callable>
struct is_function_pointer
    : std::integral_constant<bool, ...>
{
};
```

6.1.5 Is member function

Return whether the given callable type, is a member function.

```
template<typename Callable, typename... Args>
struct is_member_function
    : std::integral_constant<bool, ...>
{
};
```

Note; This currently returns false for static functions, as we're not able to disambiguate static functions from free functions. The only way to disambiguate would be a compiler supported solution.

6.1.6 Is static or free function

Return whether the given callable type, is a static or free function.

```
template<typename Callable, typename... Args>
struct is_static_member_or_free_function
    : std::integral_constant<bool, ...>
{
};
```

Note; See note in 'Is member function'. - Currently returns the complete opposite of `is_member_function`.

6.1.7 Implicit object type

Return the type of the implicit object parameter for a given member function.

```
template<typename Callable, typename... Args>
using implicit_object_type = ...;
```

Note; This is only defined for callable's returning true for; 'Is member function'

6.1.8 Parameter type

Return the type of the i'th parameter for a given callable type.

```
template<int I, typename Callable, typename... Args>
using parameter_type = ...;
```

Note; Is only defined for; $0 < I <$ `std::arity<Callable>`.
TODO; Figure if it's possible in a pure library solution, to retain qualifiers.

6.1.9 Return type

Return the return type for a given callable type.

```
template<typename Callable, typename... Args>
using return_type = ...;
```

6.2 Usage

The above traits can be used, in a few ways;

6.2.1 Functions

Using `decltype(...)`, for declared functions;

```
void function(int, double);
std::return_type<decltype(function)> // void
std::parameter_type<0, decltype(function)> // int
```

Using `decltype(&...)`, for declared functions;

```
void function(int, double);
std::return_type<decltype(&function)> // void
std::parameter_type<0, decltype(&function)> // int
```

Using template specifiction, for non-declared functions; (which the above case, maps down to after decltype)

```
std::arity<void(...)>::value // 0
std::has_varying_arguments<void(...)>::value // true
```

6.2.2 Functors

Passing functor types directly;

```
struct functor
{
    void operator()(int);
};
std::return_type<functor> // void
std::implicit_object_type<functor> // functor
```

Passing functor types directly, but specifying the overload;

```
struct variadic_template_functor
{
    template <typename... Ts>
    void operator()(Ts..., ...);
};
std::arity<variadic_template_functor, int, double>::value // 2
std::has_varying_arguments<variadic_template_functor, int, double>::value // true
std::parameter_type<1, variadic_template_functor, int, double> // double
```

6.2.3 Member functions

All class methods, are essencially function pointers, and hence will be called the same way, just with a prefix;

```
struct klass
{
    void function(int);
    void const_function(int) const;
    static void static_function(int);
};
std::is_member_function<decltype(&klass::function)>::value // true
std::is_const_function<decltype(&klass::const_function)>::value // true
std::is_static_member_or_free_function<decltype(&klass::static_function)>::value // t
```

Do note however, that for member functions (e.g. non-static functions defined on the class), we need the '`&`', while it can be left out for static functions;

```
std::is_static_member_or_free_function<decltype(klass::static_function)>::value // tr
```

The reason for why static functions are treated differently, is because these are essentially free functions, and hence subject to the C rules for functions and function pointers.

Note; Templated and overloaded methods, are handled by explicitly specifying the overload.

6.2.4 Lambdas

Lambdas are essentially functors, and hence behave as these, however as lambdas are constructed as objects of an anonymous type, we'll need `decltype(...)` to get the underlaying type.

```
const auto lambda = [](int i){ return i; };
std::is_member_function<decltype(lambda)>::value // true
std::is_const_function<decltype(lambda)>::value // true
```

Note; C++14 Polymorphic lambdas will be supported by means of explicitly specifying the overload.

6.2.5 Type overloading the call operator

For all other types, which overload the call operator (`operator()`), we have behavior as in the functor case. For instance with std::function;

```
std::is_member_function<std::function<int(int)>>::value // true
std::is_same<std::return_type<std::function<int(int)>> // int
std::is_same<std::parameter_type<0, std::function<int(int)>> // int
```

6.3 Implementatation

The proposal is concerned with the interface only, and hence compiler and standard library vendors are free to do their implementation in any way they'd like, be it compiler argumented or a pure library solution.

A pure library implementation has been developed, and is available at;

- SourceForge Repository Link (may be updated since paper release)
- Download Code as Zip (always refers to the paper release)

Note; This implementation currently makes use of a few C++1y features, however the library can be implemented without these.

7 Acknowledgements

I'd like to thanks the following people;

- 'kennytm', for his answer on StackOverflow which eventually lead me to creating this proposal.
- All the people who contributed to the discussion at 'std-proposals' (9zafJmVT2kQ), and thereby lead me to creating the second revision of this paper.