Dnnnn=yy-nnnn

Emil Madsen

Doc No:	Dnnnn=yy-nnnn
Revision:	1.1
Date:	2013-12-02
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

I Introduction

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.

The proposed interface is a trait inspired one, a listing of the proposed traits can be found in the technical specification.

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

II Motivation

Note: This following discussion on lambdas also applies to ordinary functions and methods.

When passing lambdas as function arguments or saving them to variables, one can either;

- Pass/Save the lambda as a function pointer, assuming it's capture-less.
- Pass/Save the lambda as a std::function (when wrapped), or
- Pass the lambda as a templated type / Save it using the auto keyword.

The main point of the above, is that; in order to do one of the two first, one has to explicitly specify the exact type of the lambda. While the last option saves us from doing this (which may be preferable).

However using the last method, leaves us in a somewhat unknown state, as to what the return-type, and parameter(s) for the lambda is. That is, currently there's no standard way of querying this information from the lambda variable itself / template parameter. Do please note that the same applies to functions bind via unrestricted templates or the auto keyword.

Being able to statically determine attributes of templated and auto saved lambdas at compile-time, will allow one to generate definitions, and generic code based upon these.

In essence, this means that one can write generic code, while having template specific behavior, so the issue being solved by this proposal is alike any other issue, where traits is proposed as the solution. That is in the generic environment, where assertions and/or behavior is dependent on the template arguments.

A praticta toy example of this would be the following;

```
<code>
template<typename CALLABLE>
class std_function_convert
{
    private:
        // This function doesn't need a body, as it should never be called.
        template<typename T, T... S>
        static auto convert_to_std_function(std::integer_sequence<T, S...>) ->
        std::function<typename function_traits<CALLABLE>::return_type(
            typename function_traits<CALLABLE>::template arg<S>::type...)>;
    public:
        using type = decltype(convert_to_std_function(typename
            std::make_integer_sequence<size_t,
                function_traits<CALLABLE>::arity>()));
};
</code>
```

Note: 'std::integer_sequence<T, S...>' and friends refer to the concepts from N3493. The above code allows one to convert a lambda to a std::function, by extracting the required template information from the lambda at compile time, and typedef'ing the resulting std::function inside a struct. Usage code may look like;

```
<code>
auto lambda = [](int i) { return long(i*10); };
using std_function = std_function_convert<decltype(lambda)>::type;

static_assert(std::is_same<std_function, std::function<long(int)>>::value,
              "std_function_convert is broken");
</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.
Besides from the obvious implications and usage scenarios of this, it should also be added that this proposal would be an obvious fit for the type_traits C++11 header. As it applies the traits solution, not to ordinary types, but to callable types (lambdas, function pointers and methods).

III Scope

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

IV Impact On the Standard

This proposal is a pure library extension. It does not require changes to any standard classes, functions or headers. Except if appended to the type_traits header, although a new separate header may be a great alternative, in order to refrain from changing the current headers. The suggested implementation depends on the integer_sequence, as proposed in N3493, and on std::tuple. As integer_sequence is to be included in C++14, this proposal may be candidate for that as well.

V Design Decisions

There has not been a lot of design decisions at this point, except naming and implementation, which are open to discussions obviously.

VI Technical Specifications

VI.I Interface

I'm open to adding more traits to the interface, assuming they are implementable in a pure library solution.

VI.I.1 Arity

(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>
constexpr size_t arity = ...;

// 2
template <typename Callable, typename... Args>
constexpr bool is_nullary = (arity<Callable, Args...> == 0);
template <typename Callable, typename... Args>
constexpr bool is_unary = (arity<Callable, Args...> == 1);
```

```
template <typename Callable, typename... Args>
constexpr bool is_binary = (arity<Callable, Args...> == 2);
template <typename Callable, typename... Args>
constexpr bool is_ternary = (arity<Callable, Args...> == 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'.

VI.I.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>
constexpr bool has_varying_arguments = ...;
```

Note; Has nothing to do with variadic templates.

VI.I.3 Is function pointer

Return whether the callable type is a function pointer.

```
template<typename Callable>
constexpr bool is_function_pointer = ...;
```

VI.I.4 Is const function

Return whether the given member function, is declared const.

```
template <typename Callable, typename... Args>
constexpr bool is_const_function = ...;
```

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

VI.I.5 Is member function

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

```
template<typename Callable, typename... Args>
constexpr bool is_member_function = ...;
```

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.

VI.I.6 Is static or free function

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

```
template<typename Callable, typename... Args>
constexpr bool is_static_member_or_free_function = ...;
```

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

VI.I.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'

VI.I.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.

VI.I.9 Return type

Return the return type for a given callable type.

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

VI.II Usage

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

VI.II.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(...)> // 0
std::has_varying_arguments<void(...)> // true
```

VI.II.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> // 2
std::has_varying_arguments<variadic_template_functor, int, double> // true
std::parameter_type<1, variadic_template_functor, int, double> // double
```

VI.II.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)> // true
std::is_const_function<decltype(&klass::const_function)> // true
std::is_static_member_or_free_function<decltype(&klass::static_function)> // true
```

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)> // true
```

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.

VI.II.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)> // true
std::is_const_function<decltype(lambda)> // true
```

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

VI.II.5 Type overloading 'operator()'

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

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

VI.III 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 TODO: INSERT LINK

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

VII Acknowledgements

I'd like to thanks the following people;

- 'kennytm', for his answer on StackOverflow which eventually lead me to creating this proposal.