Dnnnn=yy-nnnn

Emil Madsen

Doc No:	Dnnnn=yy-nnnn
Revision:	1.0
Date:	2013-11-06
Project:	Programming Language C++, Library Evolution Working Group
Author:	Emil Madsen
Reply to:	Emil Madsen
	Email: skeen@cs.au.dk

A Proposal to add lambda traits to the STL

Contents

I Introduction

This proposal aims to support compile-time retrieval of lambda / function return-type and parameter types, using traits. 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 pratictal 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

Possible implementation snippet, by 'kennytm', posted on StackOverflow;

```
<code>
// For generic types, forward to use signature of its 'operator()'
template<typename T>
struct function_traits
    : public function_traits<decltype(&T::operator())>
{
};

// Specialization for pointers to member functions
template<typename ClassType, typename ReturnType, typename... Args>
struct function_traits<ReturnType(ClassType::*)(Args...) const>
{
    using return_type = ReturnType;

    static const int arity = (sizeof...(Args));

    template<size_t i>
    struct arg
    {
        using type = typename
            std::tuple_element<i, std::tuple<Args...>>::type;
    };
};
</code>
```

Note; Full implementation, by 'kennytm', can be found at; GitHub.

VII Acknowledgements

I'd like to thanks 'kennytm', for his answer on StackOverflow which eventually lead me to creating this proposal.