Chapter . The Type Traits Introspection Library 1.5

Edward Diener

Table of Contents

Introduction

Welcome to the Type Traits Introspection library version 1.5.

The Type Traits Introspection library, or TTI for short, is a library of macros generating metafunctions, and a set of parallel nullary type metafunctions, which provide the ability to introspect by name the elements of a type at compile time.

The name of the library is chosen because the library offers compile time functionality on a type, similar to the Boost Type Traits library, and because the functionality the library offers is the ability to introspect a type about the existence of a specific element.

I use the word "introspect" in a very broad sense here. Normally language introspection means initially asking for information to be returned by name, which can then further be used to introspect for more specific information. In the TTI library one must always supply the name, and use the functionality for the correct type of inner element to find out if the particular named entity exists. You may prefer the term "query" instead of "introspection" to denote what this library does, but I use terminology based on the word "introspect" throughout this documentation.

The functionality of the library may be summed up as:

- Provide the means to introspect a type at compile time using a set of macros. Each macro takes the name of the type's element and generates a metafunction which can be subsequently invoked to determine whether or not the element exists within the type. These generated metafunctions will be called "macro metafunctions" in the documentation.
- Provide a set of named metafunctions which can operate on the generated macro metafunctions using a different syntax which some programmers may find easier to use. These named metafunctions will be called "nullary type metafunctions" in the documentation. This set of nullary type metafunctions can be completely ignored by programmers who do not find their syntax useful since they have the exact same functionality as the macro metafunctions above.

The library is dependent on Boost PP, Boost MPL, Boost Type Traits, and Boost Function Types. The library is also dependent on the variadic macro support of the Boost PP library currently in the sandbox if the variadic macros in the library are used.

The library is a header only library.

Since the dependencies of the library are all header only libraries, there is no need to build anything in order to use the TTI library.

Header Files

There are two separate general header files in the library, divided depending on whether or not the library functionality supporting variadic macros is to be used, which bring in all the rest of the specific header files.

1. The main header, which does not require using the library support for variadic macros, is `boost/tti/tti.hpp`. This can be used for the vast majority of functionality in the library.
2. The secondary header, which uses a small subset of the library functionality, providing an alternate use of particular macros with variadic macro support, is `boost/tti/tti_vm.hpp`.

There are also separate specific header files for each of the elements to be introspected by the library. This allows for finer-grained inclusion of nested elements to be introspected. These individual header files will be given when discussing the individual elements.

If the general header file `boost/tti/tti_vm.hpp` or the specific header file `vm_has_template_check_params.hpp`, are used, the library uses variadic macros and is also dependent on the variadic macro support of the Boost PP library currently in the Boost trunk.

Why the TTI Library ?

In the Boost Type Traits library there is compile time functionality for querying information about a C++ type. This information is very useful during template metaprogramming and forms the basis, along with the constructs of the Boost MPL library, and some other compile time libraries, for much of the template metaprogramming in Boost.

One area which is mostly missing in the Type Traits library is the ability to determine what C++ inner elements are part of a type, where the inner element may be a nested type, function or data member, static function or static data member, or class template.

There has been some of this functionality in Boost, both in already existing libraries and in libraries on which others have worked but which were never submitted for acceptance into Boost. An example with an existing Boost library is Boost MPL, where there is functionality, in the form of macros and metafunctions, to determine whether an enclosing type has a particular nested type or nested class template. An example with a library which was never submitted to Boost is the Concept Traits Library from which much of the functionality of this library, related to type traits, was taken and improved upon.

It may also be possible that some other Boost libraries, highly dependent on advanced template metaprogramming techniques, also have internal functionality to introspect a type's elements at compile time. But to the best of my knowledge this sort of functionality has never been incorporated in a single Boost library. This library is an attempt to do so, and to bring a recognizable set of interfaces to compile-time type introspection to Boost so that other metaprogramming libraries can use them for their own needs.

Terminology

The term "enclosing type" refers to the type which is being introspected. This type is always a class, struct, or union.

The term "inner xxx", where xxx is some element of the enclosing type, refers to either a type, template, function, or data within the enclosing type. The term "inner element" also refers to any one of these entities in general.

I use the term "nested type" to refer to a type within another type. I use the term "member function" or "member data" to refer to non-static functions or data that are part of the enclosing type. I use the term "static member function" or "static member data" to refer to static functions or data that are part of the enclosing type. I use the term "nested class template" to refer to a class template nested within the enclosing type.

Other terminology may be just as valid for the notion of C++ language elements within a type, but I have chosen these terms to be consistent.

The term "generated metafunction(s)" refers to macro metafunctions which are generated by macros.

The term "named metafunction(s)" refers to the specifically named nullary type metafunctions which are in the boost::tti namespace.

General Functionality

The elements about which a template metaprogrammer might be interested in finding out at compile time about a type are:

- Does it have a nested type with a particular name ?
- Does it have a nested type with a particular name which is a typedef for a particular type ?
- Does it have a nested class template with a particular name ?
- Does it have a nested class template with a particular name and a particular signature ?
- Does it have a member function with a particular name and a particular signature ?
- Does it have a member data with a particular name and of a particular type ?
- Does it have a static member function with a particular name and a particular signature ?
- Does it have a static member data with a particular name and of a particular type ?

These are the compile-time questions which the TTI library answers. It does this by creating metafunctions which can be used at compile-time.

All of the questions above attempt to find an answer about an inner element with a particular name. In order to do this using template metaprogramming, macros are used so that the name of the inner element can be passed to the macro. The macro will then generate an appropriate metafunction, which the template metaprogrammer can then use to introspect the information that is needed. The name itself of the inner element is always passed to the macro as a macro parameter, but other macro parameters may also be needed in some cases.

All of the macros start with the prefix `BOOST_TTI_`, create their metafunctions as class templates in whatever scope the user invokes the macro, and come in two forms:

1. In the simplest macro form, which I call the simple form, the 'name' of the inner element is used directly to generate the name of the metafunction as well as serving as the 'name' to introspect. In generating the name of the metafunction from the macro name, the `BOOST_TTI_` prefix is removed, the rest of the macro name is changed to lower case, and an underscore ('`_`') followed by the 'name' is appended. As an example, for the macro `BOOST_TTI_HAS_TYPE(MyType)` the name of the metafunction is `has_type_MyType` and it will look for an inner type called 'MyType'.

2. In the more complicated macro form, which I call the complex form, the macro starts with `BOOST_TTI_TRAIT_` and a 'trait' name is passed as the first parameter, with the 'name' of the inner element as the second parameter. The 'trait' name serves solely to completely name the metafunction in whatever scope the macro is invoked. As an example, for the macro `BOOST_TTI_TRAIT_HAS_TYPE(MyTrait,MyType)` the name of the metafunction is `MyTrait` and it will look for an inner type called `MyType`.

Every macro metafunction has a simple form and a corresponding complex form. Once either of these two macro forms are used for a particular type of inner element, the corresponding macro metafunction works exactly the same way and has the exact same functionality.

In the succeeding documentation all macro metafunctions will be referred by their simple form name, but remember that the complex form can alternatively always be used instead.

Macro Metafunction Name Generation

For the simple macro form, even though it is fairly easy to remember the algorithm by which the generated metafunction is named, TTI also provides, for each macro metafunction, a corresponding 'naming' macro which the end-user can use and whose sole purpose is to expand to the metafunction name. The naming macro for each macro metafunction has the form: 'corresponding-macro'_GEN(name). As an example, BOOST_TTI_HAS_TYPE(MyType) creates a metafunction which looks for a nested type called 'MyType' within some enclosing type. The name of the metafunction generated, given our algorithm above is 'has_type_MyType'. A corresponding macro called BOOST_TTI_HAS_TYPE_GEN, invoked as BOOST_TTI_HAS_TYPE_GEN(MyType) in our example, expands to the same 'has_type_MyType'. These name generating macros, for each of the metafunction generating macros, are purely a convenience for end-users who find using them easier than remembering the name-generating algorithm given above.

Macro metafunction name generation considerations

Because having a double underscore (__) in a name is reserved by the C++ implementation, creating C++ identifiers with double underscores should not avoided by the end-user. When using a TTI macro to generate a metafunction using the simple macro form, TTI appends a single underscore to the macro name preceding the name of the element that is being introspected. The reason for doing this is because Boost discourages as non=portable C++ identifiers with mixed case letters and the underscore then becomes the normal way to separate parts of an identifier name so that it looks understandable. Because of this decision to use the underscore to generate the metafunction name from the macro name, any inner element starting with an underscore will cause the identifier for the metafunction name being generated to contain a double underscore. A rule to avoid this problem is:

- When the name of the inner element to be introspected begins with an underscore, use the complex macro form, where the name of the metafunction is specifically given.

Furthermore because TTI often generates not only a metafunction for the end-user to use but some supporting detail metafunctions whose identifier, for reasons of programming clarity, is the same as the metafunction with further letters appended to it separated by an underscore, another rule is:

1. When using the complex macro form, which fully gives the name of the generated macro metafunction, that name should not end with an underscore.

Following these two simple rules will avoid names with double underscores being generated by TTI.

Reusing the named metafunction

When the end-user uses the TTI macros to generate a metafunction for introspecting an inner element of a particular type, that metafunction can be re-used with any combination of valid template parameters which involve the same type of inner element of a particular name.

As one example of this let's consider two different types called 'AType' and 'BType', for each of which we want to determine whether an inner type called 'InnerType' exists. For both these types we need only generate a single macro metafunction in the current scope by using:

BOOST_TTI_HAS_TYPE(InnerType)

We now have a metafunction, which is a C++ class template, in the current scope whose C++ identifier is 'has_type_InnerType'. We can use this same metafunction to introspect the existence of the nested type 'InnerType' in either 'AType' or 'BType' at compile time. Although the syntax for doing this has no yet been explained, I will give it here so that you can see how 'has_type_InnerType' is reused:

1. 'has_type_InnerType<AType>::value' is a compile time constant telling us whether 'InnerType' is a type which is nested within 'AType'.

2. 'has_type_InnerType<BType>::value' is a compile time constant telling us whether 'InnerType' is a type which is nested within 'BType'.

As another example of this let's consider a single type, called 'CType', for which we want to determine if it has either of two overloaded member functions with the same name of 'AMemberFunction' but with the different function signatures of 'int (int)' and 'double (long)'. For both these member functions we need only generate a single macro metafunction in the current scope by using:

BOOST_TTI_HAS_MEMBER_FUNCTION(AMemberFunction)

We now have a metafunction, which is a C++ class template, in the current scope whose C++ identifier is 'has_member_function_AMemberFunction'. We can use this same metafunction to introspect the existence of the member function 'AMemberFunction' with either the function signature of 'int (int)' or 'double (long)' in 'CType' at compile time. Although the syntax for doing this has no yet been explained, I will give it here so that you can see how 'has_type_InnerType' is reused:

1. 'has_member_function_AMemberFunction<CType,int,boost::mpl::vector<int> >::value' is a compile time constant telling us whether 'AMemberFunction' is a member function of type 'CType' whose function signtaure is 'int (int)'.

2. 'has_member_function_AMemberFunction<CType,double,boost::mpl::vector<long> >::value' is a compile time constant telling us whether 'AMemberFunction' is a member function of type 'CType' whose function signtaure is 'double (long)'.

These are just two examples of the ways a particular macro metafunction can be reused. The two 'constants' when generating a macro metafunction are the 'name' and 'type of inner element'. Once the macro metafunction for a particular name and inner element type has been generated, it can be reused for introspecting the inner element(s) of any enclosing type which correspond to that name and inner element type.

Avoiding ODR violations

The TTI macro metafunctions are generating directly in the enclosing scope in which the corresponding macro is invoked. This can be any C++ scope in which a class template can be specified. Within this enclosing scope the name of the metafunction being generated must be unique or else a C++ ODR (One Definition Rule) violation will occur. This is extremely important to remember, especially when the enclosing scope can occur in more than one translation unit, which is the case for namespaces and the global scope.

Because of ODR, and the way that the simple macro form metafunction name is directly dependent on the inner type of and name of the element being introspected, it is the responsibility of the programmer using the TTI macros to generate metafunctions to avoid ODR within a module (application or library). There are a few general methods for doing this:

1. Create unique namespace names in which to generate the macro metafunctions and/or generate the macro metafunctions in C++ scopes which can not extend across translation units. The most obvious example of this latter is within C++ classes.

2. Use the complex macro form to specifically name the metafunction generated in order to provide a unique name.

3. Avoid using the TTI macros in the global scope.

For anyone using TTI in a library which others will eventually use, it is important to generate metafunction names which are unique to that library.

TTI also reserves not only the name generated by the macro metafunction for its use but also any C++ identifier sequence which begins with that name. In other words if the metafunction being generated by TTI is named 'MyMetafunction' using the complex macro form, do not create any C++ construct with an identifier starting with 'MyMetaFunction', such as 'MyMetaFunction_Enumeration' or 'MyMetaFunctionHelper' in the same scope. All names starting wih the metafunction name in the current scope should be considered out of bounds for the programmer using TTI to use.

Macro Metafunctions

The TTI library uses macros to create metafunctions, in the current scope, for introspecting an inner element by name. Each macro for a particular type of inner element has two forms, the simple one where the first macro parameter designating the 'name' of the inner element is used to create the name of the metafunction, and the complex one where the first macro parameter, called 'trait', designates the name of the metafunction and the second macro parameter designates the 'name' to be introspected. Other than that difference, the two forms of the macro produce the exact same results.

To use these metafunctions you can include the main general header file 'boost/tti/tti.hpp', unless otherwise noted. Alternatively you can include a specific header file as given in the table below.

A table of these macros is given, based on the inner element whose existence the metaprogrammer is introspecting. More detailed explanations and examples for each of the macro metafunctions will follow this section in the documentation. The actual syntax for each macro metafunction can be found in the reference section, and examples of usage for all the macro metafunctions can be found in the "Using the Macro Metafunctions" section.

In the Template column only the name generated by the simple form of the template is given since the name generated by the complex form is always 'trait' where 'trait' is the first parameter to the corresponding complex form macro. All of the introspecting metafunctions in the table below return a boolean constant called 'value', which specifies whether or not the inner element exists.

Table 1. TTI Macro Metafunctions

Inner Element	Macro	Template	Specific Header File

Type	`BOOST_TTI_HAS_TYPE`(name)	`has_type_'name'`
		class T = enclosing type
			`has_type.hpp`

Type with check	`BOOST_TTI_HAS_TYPE`(name)	`has_type_'name'`
		class T = enclosing type
		class U = type to check against
			`has_type.hpp`

Class Template	`BOOST_TTI_HAS_TEMPLATE`(name)	`has_template_'name'`
		class T = enclosing type
		All of the template parameters must be 'class' (or 'typename') parameters
			`has_template.hpp`

Class Template with params	`BOOST_TTI_HAS_TEMPLATE_CHECK_PARAMS`(name,ppSeq[a])	`has_template_check_params_'name'`
		class T = enclosing type
			`has_template_check_params.hpp`

Class Template with params using variadic macros[b]	`BOOST_TTI_VM_HAS_TEMPLATE_CHECK_PARAMS`(name,...[c])	`has_template_check_params_'name'`
		class T = enclosing type
			`vm_has_template_check_params.hpp`

Member data	`BOOST_TTI_HAS_MEMBER_DATA`(name)	`has_member_data_'name'`
		class T = enclosing type
		class R = data type
			`has_member_data.hpp`

Member function as individual types	`BOOST_TTI_HAS_MEMBER_FUNCTION`(name)	`has_member_function_'name'`
		class T = enclosing type
		class R = return type
		class FS = (optional) function parameter types as a Boost MPL forward sequence. If there are no function parameters this does not have to be specified. Defaults to boost::mpl::vector<>.
		class TAG = (optional) Boost `function_types` tag type. Defaults to `boost::function_types::null_tag`.
			`has_member_function.hpp`

Inner Element	Macro	Template	Specific Header File
Member function as a composite type	BOOST_TTI_HAS_MEMBER_FUNCTION_WITH_SIG(name)	has_member_function_with_sig_'name' class T = pointer to member function The form for T is 'ReturnType (Class::*)(Zero or more comma-separated parameter types)'	has_member_function_with_sig.hpp

Static member data	BOOST_TTI_HAS_STATIC_MEMBER_DATA(name)	has_static_member_data_'name'
		
		class T = enclosing type
		
		class Type = data type
			has_static_member_data.hpp

Static member function as individual types	BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION(name)	has_static_member_function_'name'
		
		class T = enclosing type
		
		class R = return type
		
		class FS = (optional) function parameter types as a Boost MPL forward sequence. If there are no function parameters this does not have to be specified. Defaults to boost::mpl::vector<>.
		
		class TAG = (optional) Boost function_types tag type. Defaults to boost::function_types::null_tag.
			has_static_member_function.hpp

Static member function as a composite type	BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION_WITH_SIG(name)	has_static_member_function_with_sig_'name'
		
		class T = enclosing type
		
		class Type = function type
		
		The form for Type is 'ReturnType (Zero or more comma-separated parameter types)'
			has_static_member_function_with_sig.hpp

[a] A Boost PP data sequence with each comma separated portion of the template parameters as its own sequence element.

[b] General header file is `boost/tti/tti_vm.hpp`.

[c] The template parameters as variadic data.

Nested Types

The problem

The goal of the TTI library is never to produce a compiler error by just using the functionality in the library, whether it is invoking its function-like macros or instantiating the macro metafunctions created by them, and whether the inner element exists or not. In this sense The TTI library macros for introspecting an enclosing type for an inner element work well. But there is one exception to this general case. That exception is the crux of the discussion regarding nested types which follows.

The metafunctions generated by the TTI macros all work with types, whether in specifying an enclosing type or in specifying the type of some inner element, which may also involve types in the signature of that element, such as a parameter or return type of a function. The C++ notation for a nested type, given an enclosing type 'T' and an inner type 'InnerType', is 'T::InnerType'. If either the enclosing type 'T' does not exist, or the inner type 'InnerType' does not exist within 'T', the expression 'T::InnerType' will give a compiler error if we attempt to use it in our template instantiation of one of TTI's macro metafunctions.

We want to be able to introspect for the existence of inner elements to an enclosing type without producing compiler errors. Of course if we absolutely know what types we have and that a nested type exists, and these declarations are within our scope, we can always use an expression like T::InnerType without error. But this is often not the case when doing template programming since the type being passed to us at compile-time in a class or function template is chosen at instantiation time and is created by the user of a template.

One solution to this is afforded by the library itself. Given an enclosing type 'T' which we know must exist, either because it is a top-level type we know about or it is passed to us in some template as a 'class T' or 'typename T', and given an inner type named 'InnerType' whose existence we would like ascertain, we can use a `BOOST_TTI_HAS_TYPE(InnerType)` macro and it's related `has_type_InnerType` metafunction to determine if the nested type 'InnerType' exists. This solution is perfectly valid and, with Boost MPL's selection metafunctions, we can do compile-time selection to generate the correct template code.

However this does not ordinarily scale that well syntactically if we need to drill down further from a top-level enclosing type to a deeply nested type, or even to look for some deeply nested type's inner elements. We are going to be generating a great deal of `boost::mpl::if_` and/or `boost::mpl::eval_if` type selection statements to get to some final condition where we know we can generate the compile-time code which we want.

One solution

The TTI library offers a solution in the form of a construct which works with a nested type without producing a compiler error if the nested type does not exist, but still is able to do the introspecting for inner elements that our TTI macro metafunctions do.

This is a metafunction created by the macro `BOOST_TTI_MEMBER_TYPE`. Its general explanation is given as:

Table 2. TTI Nested Type Macro Metafunction

Inner Element	Macro	Template	Specific Header File

Type	`BOOST_TTI_MEMBER_TYPE`(name)	`member_type_'name'`
		
		class T = enclosing type
		
		returns = the type of 'name' if it exists, else an unspecified type, as the typedef 'type'.
			`member_type.hpp`

Instead of telling us whether an inner type of a particular name exists, as the metafunction generated by BOOST_TTI_HAS_TYPE does, the BOOST_TTI_MEMBER_TYPE macro generates a metafunction which, passed an enclosing type as its single template parameter, returns a typedef 'type' which is that inner type if it exists, else it is an unspecified marker 'type' if it does not. In this way we have created a metafunction, very similar in functionality to boost::mpl::identity, but which still returns some unspecified marker 'type' if our nested type is invalid.

We can use the functionality of `BOOST_TTI_MEMBER_TYPE` to construct nested types for our other macro metafunctions, without having to use the T::InnerType syntax and produce a compiler error if no such type actually exists within our scope. We can even do this in deeply nested contexts by stringing together, so to speak, a series of these macro metafunction results.

As an example, given a type T, let us create a metafunction where there is a nested type FindType whose enclosing type is eventually T, as represented by the following structure:

```
struct T
  {
  struct AType
    {
    struct BType
      {
      struct CType
        {
        struct FindType
          {
          };
        }
      };
    };
  };
```

In our TTI code we first create a series of member type macros for each of our nested types:

```
BOOST_TTI_MEMBER_TYPE(FindType)
BOOST_TTI_MEMBER_TYPE(AType)
BOOST_TTI_MEMBER_TYPE(BType)
BOOST_TTI_MEMBER_TYPE(CType)
```

Next we can create a typedef to reflect a nested type called FindType which has the relationship as specified above by instantiating our macro metafunctions.

```
typedef typename
member_type_FindType
  <
  typename member_type_CType
    <
    typename member_type_BType
      <
      typename member_type_AType
        <
        T
        >::type
      >::type
    >::type
  >::type MyFindType;
```

We can use the above typedef to pass the type as FindType to one of our macro metafunctions. FindType may not actually exist but we will not generate a compiler error when we use it, but will only generate, if it does not exist, a failure by having our meteafunction return a false value at compile-time.

As one example, let's ask whether FindType has a static member data called MyData of type 'int'. We add:

```
BOOST_TTI_HAS_STATIC_MEMBER_DATA(MyData)
```

Next we create our metafunction:

```
has_static_member_data_MyData
  <
  MyFindType,
  int
  >
```

and use this in our metaprogramming code. Our metafunction now tells us whether the nested type FindType has a static member data called MyData of type 'int', even if FindType does not actually exist as we have specified it as a type. If we had tried to do this using normal C++ nested type notation our metafunction code above would be:

```
boost::tti::has_static_member_data_MyData
  <
  typename T::AType::BType::CType::FindType,
  int
  >
```

But this fails with a compiler error if there is no such nested type, and that is exactly what we do not want in our compile-time metaprogramming code.

in the above metafunction we are asking whether or not FindType has a static member data element called 'MyData', and the result will be 'false' if either FindType does not exist or if it does exist but does not have a static member data of type 'int' called 'MyData'. In neither situation will we produce a compiler error.

Somewhere else we may also be interested in ascertaining whether the deeply nested type 'FindType' actually exists. Our metafunction, using BOOST_TTI_MEMBER_TYPE and repeating our macros from above, would be:

```
BOOST_TTI_MEMBER_TYPE(FindType)
BOOST_TTI_MEMBER_TYPE(AType)
BOOST_TTI_MEMBER_TYPE(BType)
BOOST_TTI_MEMBER_TYPE(CType)

BOOST_TTI_HAS_TYPE(FindType)

has_type_FindType
  <
  typename
  member_type_CType
    <
    typename
    member_type_BType
      <
      typename
      member_type_AType
        <
        T
        >::type
      >::type
    >::type
  >
```

But this duplicates much of our code when we generated the 'MyFindType' typedef, so it would be much better if we could simply pass our 'MyFindType' type to some other metafunction which would tell us directly if 'MyFindType' actually exists. And so we can ! The TTI library has a named metafunction in the 'boost::tti' namespace called 'valid_member_type' which takes a type and determines if it 'actually exists', returning the compile-time boolean constant called 'value' of 'true' if it does or 'false' if it does not. The meaning of 'actually exists', as far as 'boost::tti::valid_member_type' is concerned, is that the type does not equal the unspecified type which BOOST_TTI_MEMBER_TYPE returns when its nested type does not exist, and therefore 'boost::tti::valid_member_type' is meant to be used with the return 'type' of BOOST_TTI_MEMBER_TYPE, which is what 'MyFindType' represents.

The general explanation of 'boost::tti::valid_member_type' is given as:

Table 3. TTI Nested Type Macro Metafunction Existence

Inner Element	Macro	Template	Specific Header File
Type	None	boost::tti::valid_member_type class T = a type returns = true if the type exists, false if it does not. 'Existence' is determined by whether the type does not equal an unspecified type.	member_type.hpp

Using this functionality with our 'MyFindType' type above we could create the nullary metafunction:

```
boost::tti::valid_member_type
  <
  MyFindType
  >
```

directly instead of replicating the same functionality with our 'boost::tti::has_type_FindType' metafunction.

Nested Types and Function Signatures

The strength of `BOOST_TTI_MEMBER_TYPE` to represent a type which may or may not exist, and which then can be subsequently used in other macro metafunctions whenever a type is needed as a template parameter, without producing a compiler error, should not be underestimated. It is one of the reasons why we have two different versions of metafunctions for introspecting a member function or static member function of a type.

In the more general case, when using `BOOST_TTI_HAS_MEMBER_FUNCTION_WITH_SIG` and `BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION_WITH_SIG`, the signature for the member function and the static member function is a composite type. This makes for a syntactical notation which is natural to specify, but because of that composite type notation we can not use the nested type functionality in `BOOST_TTI_MEMBER_TYPE` very easily.

But in the more specific case, when we use `BOOST_TTI_HAS_MEMBER_FUNCTION` and `BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION`, the composite types in our signatures are broken down into their individual types so that using `BOOST_TTI_MEMBER_TYPE`, if necessary, for any one of the individual types is easy and natural.

Using the Macro Metafunctions

Using the macro metafunctions can be illustrated by first creating some hypothetical user-defined type with corresponding nested types and other inner elements. With this type we can illustrate the use of the macro metafunctions. This is just meant to serve as a model for what a type T might entail from within a class or function template.

```
// An enclosing type

struct AType
  {

  // Type

  typedef int AnIntType; // as a typedef

  struct BType // as a nested type
    {
    struct CType
      {
      };
    };

  // Template

  template <class> struct AMemberTemplate { };
  template <class,class,int,class,template <class> class InnerTemplate,class,long> struct ManyPara↵
meters { };
  template <class,class,int,short,class,template <class,int> class InnerTemplate,class> struct More↵
Parameters { };

  // Data

  BType IntBT;

  // Function

  int IntFunction(short) { return 0; }

  // Static Data

  static short DSMember;

  // Static Function

  static int SIntFunction(long,double) { return 2; }

  };
```

I will be using the type above just to illustrate the sort of metaprogramming questions we can ask of some type T which is passed to the template programmer in a class template. Here is what the class template might look like:

```
#include <boost/tti/tti.hpp>

template<class T>
struct OurTemplateClass
  {

  // compile-time template code regarding T

  };
```

Now let us create and invoke the macro metafunctions for each of our inner element types, to see if type T above corresponds to our hypothetical type above. Imagine this being within 'OurTemplateClass' above. In the examples below the same macro is invoked just once to avoid ODR violations.

Type

Does T have a nested type called 'AnIntType' ?

```
BOOST_TTI_HAS_TYPE(AnIntType)

boost::tti::has_type_AnIntType
  <
  T
  >
```

Does T have a nested type called 'BType' ?

```
BOOST_TTI_HAS_TYPE(BType)

boost::tti::has_type_BType
  <
  T
  >
```

Type checking the typedef

Does T have a nested typedef called 'AnIntType' whose type is an 'int' ?

```
boost::tti::has_type_AnIntType
  <
  T,
  int
  >
```

Template

Does T have a nested class template called 'AMemberTemplate' whose template parameters are all types ('class' or 'typename') ?

```
BOOST_TTI_HAS_TEMPLATE(AMemberTemplate)

boost::tti::has_template_AMemberTemplate
  <
  T
  >
```

Template with params

Does T have a nested class template called 'MoreParameters' whose template parameters are specified exactly ?

```
BOOST_TTI_HAS_TEMPLATE_CHECK_PARAMS(MoreParameters,(class)(class)(int)(short)(class)(tem↵
plate <class)(int> class InnerTemplate)(class))

boost::tti::has_template_check_params_MoreParameters
  <
  T
  >
```

Template with params using variadic macros

[note Include the `boost/tti/tti_vm.hpp` general header file, or the `vm_template_params.hpp` specific header file, when using this macro.]

Does T have a nested class template called 'ManyParameters' whose template parameters are specified exactly ?

```
BOOST_TTI_VM_HAS_TEMPLATE_CHECK_PARAMS(ManyParameters,class,class,int,class,tem↵
plate <class> class InnerTemplate,class,long)

boost::tti::has_template_check_params_ManyParameters
  <
  T
  >
```

Member data

Does T have a member data called 'IntBT' whose type is 'AType::BType' ?

```
BOOST_TTI_HAS_MEMBER_DATA(IntBT)

boost::tti::has_member_data_IntBT
  <
  T,
  AType::BType
  >
```

Member function with individual types

Does T have a member function called 'IntFunction' whose type is 'int (short)' ?

```
BOOST_TTI_HAS_MEMBER_FUNCTION(IntFunction)

boost::tti::has_member_function_IntFunction
  <
  T,
  int,
  boost::mpl::vector<short>
  >
```

Member function with composite type

Does T have a member function called 'IntFunction' whose type is 'int (short)' ?

```
BOOST_TTI_HAS_MEMBER_FUNCTION_WITH_SIG(IntFunction)

boost::tti::has_comp_member_function_IntFunction
  <
  int (T::*)(short)
  >
```

Static member data

Does T have a static member data called 'DSMember' whose type is 'short' ?

```
BOOST_TTI_HAS_STATIC_MEMBER_DATA(DSMember)

boost::tti::has_static_member_data_DSMember
  <
  T,
  short
  >
```

Static member function with individual types

Does T have a static member function called 'SIntFunction' whose type is 'int (long,double)' ?

```
BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION(SIntFunction)

boost::tti::has_static_member_function_SIntFunction
  <
  T,
  int,
  boost::mpl::vector<long,double>
  >
```

Static member function with composite type

Does T have a static member function called 'SIntFunction' whose type is 'int (long,double)' ?

```
BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION_WITH_SIG(SIntFunction)

boost::tti::has_comp_static_member_function_SIntFunction
  <
  T,
  int (long,double)
  >
```

Member type

Create a nested type T::BType::CType without creating a compiler error if T does not have the nested type BType::CType ?

```
BOOST_TTI_MEMBER_TYPE(BType)
BOOST_TTI_MEMBER_TYPE(CType)

typename
boost::tti::member_type_CType
  <
  typename
  boost::tti::member_type_BType
    <
    T
    >::type
  >::type
```

Member type existence

Does a nested type T::BType::CType, created without creating a compiler error if T does not have the nested type BType::CType, actually exist ?

```
BOOST_TTI_MEMBER_TYPE(BType)
BOOST_TTI_MEMBER_TYPE(CType)

typedef typename
boost::tti::member_type_CType
  <
  typename
  boost::tti::member_type_BType
    <
    T
    >::type
  >::type
AType;

boost::tti::valid_member_type
  <
  AType
  >
```

Macro Metafunctions as Metadata

As specified in the Boost MPL library, there are two ways to pass metafunctions as metadata, and both ways fall under the Boost MPL terminology of 'lambda expressions':

- As a metafunction class
- As a placeholder expression

Using a placeholder expression is the easiest way and does not require the programmer to create a metafunction class for passing the metadata. The syntax for this is fairly simple. The syntax for passing a macro metafunction becomes `macrometafunction<_>` etc. depending on how many parameters are being passed. Thus for two parameters we would have `macrometafunction<_,_>` etc., with another placeholder (_) added for each subsequent parameter.

The Boost MPL library also provides the high-level primitive 'quote' which creates a metafunction class from any n-ary metafunction. The syntax for this is also fairly easy. The syntax for passing a macro metafunction as a metafunction class becomes `boost::mpl::quote(n)<macrometafunction>' where 'n' depends on the actual number of template parameters the macro metafunction has.

As will be seen when we discuss the nullary type metafunctions, each nullary type metafunction takes as its first parameter a macro metafunction as metadata in the form of a lambda expression. Therefore we can use either a metafunction class generated by boost::mpl::quote(n), or a placeholder expression, to pass a macro metafunction as metadata to our nullary type metafunctions.

Nullary Type Metafunctions

The macro metafunctions provide a complete set of functionality for anything one would like to do using the TTI library.

What is presented here is a set of named metafunctions, in the boost::tti namespace, which offer an alternative syntax for using the macro metafunctions. No new functionality is involved.

The macro metafunctions, like all metafunctions, pass their parameters as types. When passing the result of one metafunction as a parameter to another metafunction, one is reaching into the first metafunction to access the nested 'type' of its instantiated result in order to pass it to another metafunction as a parameter. This is common metafunction usage, and is the syntax of the macro metafunctions as well.

An alternative use of metafunctions is to pass the instantiated metafunction itself as a nullary metafunction, rather than its nested 'type' member, as a parameter to another metafunction. This is purely a syntactical advantage of not having to manually specify 'typename' and '::type' in order to extract the nested 'type' from the resulting nullary metafunction.

This what these named metafunctions do, and therefore they are differentiated from the macro metafunctions by the name 'nullary type metafunctions'. The nullary type metafunctions reuse the macro metafunctions but pass other parameters which are classes as nullary metafunctions. While the syntactical advantage over the macro metafunctions is that one does not have to use 'typename' and pass the actual '::type' of a resultant metafunction instantiation, the disadvantage is that for class types which are not metafunction results one has to create a nullary metafunction by wrapping the class type with boost::mpl::identity.

Because the nullary type metafunctions reuse the metafunctions generated by the macro metafunctions, the metafunction generated by a particular macro metafunction needs to be passed as metadata to a corresponding nullary type metafunction. As explained in Boost MPL, a metafunction as metadata is passed as a lambda expression, in the form of a metafunction class or a placeholder expression. This lambda expression is passed as the first parameter to each of our nullary type metafunctions.

The exceptions to the use of nullary type metafunctions when specifying class 'types' are:

1. When a Boost `function_types` tag type, which is optional, is specified as an addition to the function signature it is passed as is.

2. When specifying a function signature and parameter types being passed, the MPL forward sequence which contains the parameter 'types' is passed as is.

Whenever any type is not a class, it can passed to the nullary type metafunctions as is or it can be passed as a nullary metafunction by wrapping it with boost::mpl::identity. The nullary type metafunctions handle both situations but clearly it is easier to pass a known non-class type as is than having to wrap it with boost::mpl::identity.

For nested types, which may or may not exist, we can pass the resulting nullary metafunction generated by `BOOST_TTI_MEMBER_TYPE`, raher than its nested ::type, or its equivalent nullary type metafunction `boost::tti::mf_member_type` (explained later).

To use these metafunctions you need to include the main header file `boost/tti/tti.hpp`, unless otherwise noted. Alternatively you can include a specific header as given in the table below,

Tip

The header files <boost/mpl/identity.hpp>, <boost/mpl/placeholders.hpp>, and <boost/mpl/quote.hpp> are included by the TTI header files whenever you include a general header file or a specific header file for a nullary type metafunction. Also the header file <boost/mpl/vector.hpp> is included by the general header file 'boost/tti/tti.hpp' or the specific header files which introspect functions, so if you use an MPL vector as your forward sequence wrapper for parameter types, you need not manually include the header file.

A table of the nullary type metafunctions is given, based on the inner element whose existence the metaprogrammer is introspecting. The arguments to the nullary type metafunctions are the same as those of their equivalent macro metafunction, which is passed as the first argument in the form of a Boost MPL lambda expression.

The actual syntax for each nullary type metafunction can be found in the reference section, and general examples of usage can be found in the "Using the Nullary Type Metafunctions" section.

All of the metafunctions are in the top-level 'boost::tti' namespace, all have a particular name based on the type of its functionality, and all begin with the prefix 'mf_', representing the fact that they are metafunctions and to distinguish them from any other constructs created by TTI in the boost::tti namespace.

Table 4. TTI Nullary Type Metafunctions

Inner Element	Template	Parameters	Macro Metafunction Passed	Specific Header File
Type	boost::tti::mf_has_type	class HasType = macro metafunction as lambda expression class T = enclosing type nullary metafunction	BOOST_TTI_HAS_TYPE	mf_has_type.hpp
Type with check	boost::tti::mf_has_type	class HasType = macro metafunction as lambda expression class T = enclosing type nullary metafunction class U = type to check against nullary metafunction	BOOST_TTI_HAS_TYPE	mf_has_type.hpp
Class Template	boost::tti::mf_has_template	class HasTemplate = macro metafunction as lambda expression class T = enclosing type nullary metafunction	BOOST_TTI_HAS_TEMPLATE	mf_has_template.hpp
Class Template with params	boost::tti::mf_has_template_check_params	class HasTemplateCheckParams = macro metafunction as lambda expression class T = enclosing type nullary metafunction	BOOST_TTI_HAS_TEMPLATE_CHECK_PARAMS BOOST_TTI_VM_HAS_TEMPLATE_CHECK_PARAMS	mf_has_template_check_params.hpp
Member data	boost::tti::mf_has_member_data	class HasMemberData = macro metafunction as lambda expression class T = enclosing type nullary metafunction class R = type of member data nullary Metafunction	BOOST_TTI_HAS_MEMBER_DATA	mf_has_member_data.hpp

Inner Element	Template	Parameters	Macro Metafunction Passed	Specific Header File
Member function	`boost::tti::mf_has_member_function`	class HasMemberFunction = macro metafunction as lambda expression class T = enclosing type nullary metafunction class R = return value nullary metafunction class FS = (optional) a Boost MPL forward sequence of parameter types as nullary metafunctions. The forward sequence as a type is not presented as a nullary metafunction. If there are no parameters, this may be omitted. class TAG = (optional) a Boost `function_types` tag type.	`BOOST_TTI_HAS_MEMBER_FUNCTION`	`mf_has_member_function.hpp`
Static data	`boost::tti::mf_has_static_member_data`	class HasStaticMemberData = macro metafunction as lambda expression class T = enclosing type nullary metafunction class R = type of static data nullary metafunction	`BOOST_TTI_HAS_STATIC_MEMBER_DATA`	`mf_has_static_member_data.hpp`

Inner Element	Template	Parameters	Macro Metafunction Passed	Specific Header File
Static function	boost::tti::mf_has_static_member_function	class HasStaticMemberFunction = macro metafunction as lambda expression class T = enclosing type nullary metafunction class R = return value nullary metafunction class FS = (optional) a Boost MPL forward sequence of parameter types as nullary metafunctions. The forward sequence as a type is not presented as a nullary metafunction. If there are no parameters, this may be omitted. class TAG = (optional) a Boost function_types tag type.	BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION	mf_has_static_member_function.hpp

The nullary type metafunctions have no equivalent to the macro metafunctions which work with function types in their composite form. These are the macro metafunctions generated by the BOOST_TTI_MEMBER_FUNCTION_WITH_SIG and BOOST_TTI_STATIC_MEMBER_FUNCTION_WITH_SIG macros. The reason for this is that the nullary type metafunctions work with individual types, and the ability to pass those individual types as nullary functions when they are class types.

Nullary type metafunction `member_type` equivalent

Just as there exists the macro `BOOST_TTI_MEMBER_TYPE` for creating a macro metafunction which returns a nested type if it exists, else an unspecified marker type, there is also the equivalent nullary type metafunction.

Table 5. TTI Nested Type Nullary Type Metafunction

Inner Element	Template	Parameters	Macro Metafunction Passed	Specific Header File
Type	boost::tti::mf_member_type	class MemberType = macro metafunction as lambda expression class T = enclosing type nullary metafunction	BOOST_TTI_MEMBER_TYPE	mf_member_type.hpp

The difference between the macro metafunction `BOOST_TTI_MEMBER_TYPE` and `boost::tti::mf_member_type` is simply that, like the other nullary type metafunctions, the latter takes its enclosing type as a nullary metafunction. Both produce the exact same result.

Nullary type metafunction `valid_member_type` equivalent

Also similar to the macro metafunctions, we have an easy way of testing whether or not our `boost::tti::mf_member_type` nested type actually exists.

Table 6. TTI Nested Type Nullary Type Metafunction Existence

Inner Element	Template	Parameters	Specific Header File
Type	`boost::tti::mf_valid_member_type`	class T = a type as a nullary metafunction returns = true if the nullary metafunction's inner 'type' exists, false if it does not. 'Existence' is determined by whether the type does not equal an unspecified type.	`mf_member_type.hpp`

Note the difference here from the equivalent macro metafunction tester `boost::tti::valid_member_type`. In the table above the type T is passed as a nullary metafunction holding the actual type, where for the macro metafunction equivalent the type T is passed as the actual type being tested.

Using the Nullary Type Metafunctions

Using the nullary type metafunctions can be illustrated by first creating some hypothetical user-defined type with corresponding nested types and other inner elements. This user-defined type will be weighted toward showing deeply nested types and elements within those nested types. With this type we can illustrate the use of the nullary type metafunctions. This is just meant to serve as a model for what a type T might entail from within a class or function template.

```
// An enclosing type

struct T
  {

  // Type

  struct BType // as a nested type
    {

    // Template

    template <class,class,int,class,template <class> class InnerTem↵
plate,class,long> struct ManyParameters { };

    // Type

    struct CType // as a further nested type
      {

      // Template

      template <class> struct AMemberTemplate { };

      // Type

      struct DType // as a futher nested type
        {

        // Type

        typedef double ADoubleType;

        // Template

        template <class,class,int,short,class,template <class,int> class InnerTem↵
plate,class> struct MoreParameters { };

        // Function

        int IntFunction(short) const { return 0; }

        // Static Data

        static short DSMember;

        // Static Function

        static int SIntFunction(long,double) { return 2; }

        };
      };
    };
  // Data

  BType IntBT;

  };
```

I will be using the type above just to illustrate the sort of metaprogramming questions we can ask of some type T which is passed to the template programmer in a class template. Here is what the class template might look like:

```
#include <boost/tti/tti.hpp>

template<class T>
struct OurTemplateClass
  {

  // compile-time template code regarding T

  };
```

Now let us create and invoke the nested metafunctions for each of our inner element types, to see if type T above corresponds to our hypothetical type above. Imagine this being within 'OurTemplateClass' above. In the examples below the same macro is invoked just once to avoid ODR violations.

I will also be mixing the way the macro metafunction metadata is passed to our nullary type metafunctions, whether as a metafunction class or as a placeholder expression. Both will work just fine since our nullary type metafunctions work with any lambda expression as the first template parameter.

Member type

We start off by creating typedef's, as nullary metafunctions, for our theoretical inner types in relation to T . None of these typedefs will produce a compiler error even if our structure does not correspond to T's reality. This also illustrates using 'boost::tti::mf_member_type'.

```
BOOST_TTI_MEMBER_TYPE(BType)
BOOST_TTI_MEMBER_TYPE(CType)
BOOST_TTI_MEMBER_TYPE(DType)

typedef
boost::tti::mf_member_type
  <
  member_type_BType<_>,
  boost::mpl::identity<T>
  >
BTypeNM;

typedef
boost::tti::mf_member_type
  <
  member_type_CType<_>,
  BTypeNM
  >
CTypeNM;

typedef
boost::tti::mf_member_type
  <
  member_type_DType<_>,
  CTypeNM
  >
DTypeNM;
```

We will use these typedefs in the ensuing examples.

Type

Does T have a nested type called 'DType' within 'BType::CType' ?

```
BOOST_TTI_HAS_TYPE(DType)

boost::tti::mf_has_type
  <
  has_type_DType<_>,
  CTypeNM
  >
```

We could just have easily used the `boost::tti::mf_valid_member_type` metafunction to the same effect:

```
boost::tti::mf_valid_member_type
  <
  DTypeNM
  >
```

Type checking the typedef

Does T have a nested typedef called 'ADoubleType' within 'BType::CType::DType' whose type is a 'double' ?

```
BOOST_TTI_HAS_TYPE(ADoubleType)

boost::tti::mf_has_type
  <
  has_type_ADoubleType<_>,
  DTypeNM,
  double
  >
```

Template

Does T have a nested class template called 'AMemberTemplate' within 'BType::CType' whose template parameters are all types ('class' or 'typename') ?

```
BOOST_TTI_HAS_TEMPLATE(AMemberTemplate)

boost::tti::mf_has_template
  <
  has_template_AMemberTemplate<_>,
  CTypeNM
  >
```

Template with params

Does T have a nested class template called 'ManyParameters' within 'BType' whose template parameters are specified exactly ?

```
BOOST_TTI_HAS_TEMPLATE_CHECK_PARAMS(ManyParameters,(class)(class)(int)(class)(tem↵
plate <class> class InnerTemplate)(class)(long))

boost::tti::mf_has_template_check_params
  <
  has_template_check_params_ManyParameters<_>,
  BTypeNM
  >
```

Template with params using variadic macros

Does T have a nested class template called 'MoreParameters' within 'BType::CType' whose template parameters are specified exactly ?

[note Include the `boost/tti/tti_vm.hpp` general header file, or the `boost/tti/vm_has_template_check_params.hpp` specific header file, when using this macro.]

```
BOOST_TTI_VM_HAS_TEMPLATE_CHECK_PARAMS(MoreParameters,class,class,int,short,class,tem↵
plate <class,int> class InnerTemplate,class)

boost::tti::mf_has_template_check_params
  <
  has_template_check_params_MoreParameters<_>,
  CTypeNM
  >
```

Member data

Does T have a member data called 'IntBT' whose type is 'BType' ?

```
BOOST_TTI_HAS_MEMBER_DATA(IntBT)

boost::tti::mf_has_member_data
  <
  has_member_data_IntBT<_,_>,
  boost::mpl::identity<T>,
  BTypeNM
  >
```

Member function

Does T have a member function called 'IntFunction' within 'BType::CType::DType' whose type is 'int (short) const' ?

```
BOOST_TTI_HAS_MEMBER_FUNCTION(IntFunction)

boost::tti::mf_has_member_function
  <
  has_member_function_IntFunction<_,_,_,_>,
  DTypeNM,
  int,
  boost::mpl::vector<short>,
  boost::function_types::const_qualified
  >
```

Static member data

Does T have a static member data called 'DSMember' within 'BType::CType::DType' whose type is 'short' ?

```
BOOST_TTI_HAS_STATIC_MEMBER_DATA(DSMember)

boost::tti::mf_has_static_member_data
  <
  has_static_member_data_DSMember<_._>,
  DTypeNM,
  short
  >
```

Static member function

Does T have a static member function called 'SIntFunction' within 'BType::CType::DType' whose type is 'int (long,double)' ?

```
BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION(SIntFunction)

boost::tti::mf_has_static_member_function
  <
  has_static_member_function_SIntFunction<_,_,_>,
  DTypeNM,
  int,
  boost::mpl::vector<long,double>
  >
```

Introspecting Function Templates

The one nested element which the TTI library does not introspect is function templates.

Function templates, like functions, can be member function templates or static member function templates. In this respect they are related to functions. Function templates represent a family of possible functions. In this respect they are similar to class templates, which represent a family of possible class types.

The technique for introspecting class templates in the TTI library is taken from the implementation of the technique in the Boost MPL library. In the case of `BOOST_TTI_HAS_TEMPLATE` it directly uses the Boost MPL library functionality while in the case of `BOOST_TTI_HAS_TEMPLATE_CHECK_PARAMS` it replicates much of the technique in the Boost MPL library. The technique depends directly on the fact that in C++ we can pass a template as a parameter to another template using what is called a "template template" parameter type.

One obvious thing about a template template parameter type is that it is a class template. For whatever historical or technical reasons, no one has ever proposed that C++ have a way of passing a function template directly as a template parameter, perhaps to be called a "function template template" parameter type. I personally think this would be a good addition to C++ and would make the ability of passing a template as a parameter to another template more orthogonal, since both class templates and function templates would be supported. My efforts to discuss this on the major C++ newsgroups have met with arguments both against its practical usage and the justification that one can pass a function template to another template nested in a non-template class. But of course we can do the same thing with class templates, which is in fact what Boost MPL does to pass templates as metadata, yet we still have template template parameters as class templates.

Nonetheless the fact that we can pass class templates as a template parameter but not function templates as a template parameter is the major factor why there is no really good method for introspecting function templates at compile time.

Instantiating a nested function template

There is, however, an alternate but less certain way of introspecting a function template. I will endeavor to explain why this way is not currently included in the TTI library, but first I will explain what it is.

It is possible to check whether some particular instantiation of a nested function template exists at compile-time without generating a compiler error. While this does not prove that the nested function template does not exist, since the instantiation itself may be incorrect and fail even when the nested function template exists, it provides a partially flawed means of checking.

The code to do this for member function templates looks like this (similar code also exists for static member function templates):

```
template
  <
  class C,
  class T
  >
struct TestFunctionTemplate
  {
  typedef char Bad;
  struct Good { char x[2]; };
  template<T> struct helper;
  template<class U> static Good check(helper<&U::template SomeFuncTemplate↵
Name<int,long,double> > *);
  template<class U> static Bad check(...);
  static const bool value=sizeof(check<C>(0))==sizeof(Good);
  };
```

where 'SomeFuncTemplateName' is the name of the nested function template, followed by some parameters to instantiate it. The 'class T' is the type of the instantiated member function template as a member function, and 'class C' is the type of the enclosing class.

As an example if we had:

```
struct AType
  {
  template<class X,class Y,class Z> double SomeFuncTemplateName(X,Y *,Z &) { return 0.0; }
  };
```

then instantiating the above template with:

```
TestFunctionTemplate
  <
  AType,
  double (AType::*)(int,long *,double &)
  >
```

would provide a compile-time boolean value which would tell us whether the nested member function template exists for the particular instantiation provided above. Furthermore, through the use of a macro, the TTI library could provide the means for specifying the name of the nested member function template ('SomeFuncTemplateName' above) and its set of instantiated parameters ('int,long,double' above) for generating the template.

So why does not the TTI library not provide at least this much functionality for introspecting member function templates, even if it represents a partially flawed way of doing so ?

The reason is stunningly disappointing. Although the above code is perfectly correct C++ code ('clang' works correctly), two of the major C++ compilers, in all of their different releases, can not handle the above code correctly. Both gcc (g++) and Visual C++ incorrectly choose the wrong 'check' function even when the correct 'check' function applies (Comeau C++ also fails but I am less concerned about that compiler since it is not used nearly as much as the other two). All my attempts at alternatives to the above code have also failed. The problems with both compilers, in this regard, can be seen more easily with this snippet:

```
struct AType
 {
 template<class AA> void SomeFuncTemplate() { }
 };

template<class T>
struct Test
 {
 template<T> struct helper;
 template<class U> static void check(helper<&U::template SomeFuncTemplate<int> > *) { }
 };

int main()
  {
  Test< void (AType::*)() >::check<AType>(0);
  return 0;
  }
```

Both compilers report compile errors with this perfectly correct code,

gcc:

```
error: no matching function for call to 'Test<void (AType::*)()>::check(int)'
```

and msvc:

```
error C2770: invalid explicit template argument(s) for 'void Test<T>::check(Test<T>::help↵
er<&U::SomeFuncTemplate<int>> *)'
```

There is a workaround for these compiler problems, which is to hardcode the name of the enclosing class, via a macro, in the generated template rather than pass it as a template type. In that case both compilers can handle both the member function code and the code snippet above correctly. In essence, when the line:

```
template<class U> static void check(helper<&U::template SomeFuncTemplate<int> > *) { }
```

gets replaced by:

```
template<class U> static void check(helper<&AType::template SomeFuncTemplate<int> > *) { }
```

both gcc and Visual C++ work correctly. The same goes for the 'check' line in the 'TestFunctionTemplate' above.

But the workaround destroys one of the basic tenets of the TTI library, which is that the enclosing class be passed as a template parameter, especially as the enclosing class need not actually exist (see `BOOST_TTI_MEMBER_TYPE` and the previous discussion of 'Nested Types'), without producing a compiler error. So I have decided not to implement even this methodology to introspect nested function templates in the TTI library.

Testing TTI

In the libs/tti/test subdirectory there is a jamfile which can be used to test TTI functionality.

Executing the jamfile without a target will run tests for both basic TTI and for the variadic macro portion of TTI. To successfully do that you need to get the `variadic_macro_data` library from the sandbox. You can run tests for only the basic TTI, which is the vast majority of TTI functionality, by specifying only the 'tti' target when executing the jamfile, and therefore you would not need the `variadic_macro_data` library. If you just want to run the tests for the variadic macro portion of TTI, specify the target as 'ttivm'.

The TTI library has been tested with VC++ 8, 9, 10 and with gcc 3.4.2, 3.4.5, 4.3.0, 4.4.0, 4.5.0-1, and 4.5.2-1.

History

Version 1.5

- The TTI has been accepted into Boost. This is the first iteration of changes as the library is being prepared for Boost based on the library review and end-user comments and suggestions. For each iteration of changes made based on end-user comments and suggestions, I will produce a new version number so that end-users who want to follow the progress of the library for Boost can know what is being changed. I will be targeting Boost 1.49 for completing all changes and passing all tests in order to have TTI ready to be copied from Boost trunk to Boost release for inclusion into Boost.
- Breaking changes
    - Macro metafunctions are no longer generated in any namespace, but directly in the scope of the end-user.
    - The metafunction class generating macros, for each metafunction macro, have been removed. The end-user can use boost::mpl::quote instead if he wishes.
    - The metafunction name generating macros have been simplified so that no namespace name is generated, and for each macro of the macro metafunctions there is a single metafunction name generating macro.
    - The BOOST_TTI_TRAITS_GEN macro has been removed.
    - Individual header file names have changed to more closely reflect the metafunction macro names.
    - The names of the composite member function and composite static member function macros have changed from BOOST_TTI_HAS_COMP_MEMBER_FUNCTION to BOOST_TTI_HAS_MEMBER_FUNCTION_WITH_SIG and from BOOST_TTI_HAS_COMP_STATIC_MEMBER_FUNCTION to BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION_WITH_SIG.
- All template parameter names are now unique to TTI to avoid name clashes.
- Nullary type metafunctions can be passed non-class types as is.

Version 1.4

- Breaking changes
    - `BOOST_TTI_HAS_MEMBER` (`BOOST_TTI_TRAIT_HAS_MEMBER`) has been changed to `BOOST_TTI_HAS_COMP_MEMBER_FUNCTION` (`BOOST_TTI_TRAIT_HAS_COMP_MEMBER_FUNCTION`) and `BOOST_TTI_MTFC_HAS_MEMBER` (`BOOST_TTI_MTFC_TRAIT_HAS_MEMBER`) has been changed to `BOOST_TTI_MTFC_HAS_COMP_MEMBER_FUNCTION` (`BOOST_TTI_MTFC_TRAIT_HAS_COMP_MEMBER_FUNCTION`). This family of functionality now supports only member functions with composite syntax.
    - `BOOST_TTI_HAS_STATIC_MEMBER` (`BOOST_TTI_TRAIT_HAS_STATIC_MEMBER`) has been changed to `BOOST_TTI_HAS_COMP_STATIC_MEMBER_FUNCTION` (`BOOST_TTI_TRAIT_HAS_COMP_STATIC_MEMBER_FUNCTION`) and `BOOST_TTI_MTFC_HAS_STATIC_MEMBER` (`BOOST_TTI_MTFC_TRAIT_HAS_STATIC_MEMBER`) has been changed to `BOOST_TTI_MTFC_HAS_COMP_STATIC_MEMBER_FUNCTION` (`BOOST_TTI_MTFC_TRAIT_HAS_COMP_STATIC_MEMBER_FUNCTION`). This family of functionality now supports only static member functions with composite syntax.
    - `boost::tti::mf_has_static_data` has been changed to `boost::tti::mf_has_static_member_data`.
- Added `BOOST_TTI_HAS_STATIC_MEMBER_DATA` and family for introspecting static member data.
- Inclusion of specific header files for faster compilation is now supported.
- Inclusion of macro metafunction name generating macros.
- Shorten the names of the test files and test header files.

- Added documentation topic about introspecting function templates.

Version 1.3

- Breaking changes
  - The names of the main header files are shortened to 'boost/tti/tti.hpp' and 'boost/tti/tti_vm.hpp'.
  - The library follows the Boost conventions.
    - Changed the filenames to lower case and underscores.
    - The top-level tti namespace has become the boost::tti namespace.
    - The macros now start with `BOOST_TTI_` rather than just `TTI_` as previously.
  - The variadic macro support works only with the latest version of the variadic_macro_library, which is version 1.3+.

Version 1.2

- Added the set of metafunction class macros for passing the macro metafunctions as metadata. This complements passing the macro metafunctions as metadata using placeholder expressions.

Version 1.1

- Library now also compiles with gcc 3.4.2 and gcc 3.4.5.
- Examples of use have been added to the documentation.
- In the documentation the previously mentioned 'nested type metafunctions' are now called "nullary type metafunctions'.
- `BOOST_TTI_HAS_TYPE` and `boost::tti::mf_has_type` now have optional typedef checking.
- New macro metafunction functionality which allows composite typed to be treated as individual types has been implemented. These include:
  - `BOOST_TTI_HAS_MEMBER_DATA`
  - `BOOST_TTI_HAS_MEMBER_FUNCTION`
  - `BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION`
- New nullary type metafunction `boost::tti::mf_has_static_member_function` uses the new underlying `BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION` macro metafunction. Its signature uses an optional MPL forward sequence for the parameter types and an optional Boost `function_types` tag type.
- New nullary type metafunctions `boost::tti::valid_member_type` and `boost::tti::mf_valid_member_type` for checking if the 'type' returned from invoking the `BOOST_TTI_MEMBER_TYPE` or `boost::tti::mf_member_type` metafunctions is valid.
- Breaking changes
  - `BOOST_TTI_HAS_TYPE_CHECK_TYPEDEF` and `boost::tti::mf_has_type_check_typedef` have been removed, and the functionality in them folded into `BOOST_TTI_HAS_TYPE` and `boost::tti::mf_has_type`.
  - BOOST_TTI_MEMBER_TYPE and boost::tti::mf_member_type no longer also return a 'valid' boolean constant. Use boost::tti::valid_member_type or boost::tti::mf_valid_member_type metafunctions instead (see above).
  - `boost::tti::mf_has_static_function` has been removed and its functionality moved to `boost::tti::mf_has_static_member_function` (see above).

- `boost::tti::mf_member_data` uses the new underlying `BOOST_TTI_HAS_MEMBER_DATA` macro metafunction.
- The signature for `boost::tti::mf_has_member_function` has changed to use an optional MPL forward sequence for the parameter types and an optional Boost `function_types` tag type.
- All nullary type metafunctions take their corresponding macro metafunction parameter as a class in the form of a Boost MPL lambda expression instead of as a template template parameter as previously. Using a placeholder expression is the easiest way to pass the corresponding macro metafunction to its nullary type metafunction.

Version 1.0

Initial version of the library.

ToDo

- Improve tests
- Improve documentation

Acknowledgments

The TTI library came out of my effort to take the `type_traits_ext` part of the unfinished Concept Traits Library and expand it. So my first thanks go to Terje Slettebo and Tobias Schwinger, the authors of the CTL. I have taken, and hopefully improved upon, the ideas and implementation in that library, and added some new functionality.

I would also like to thank Joel Falcou for his help and his introspection work.

Two of the introspection templates are taken from the MPL and lifted into my library under a different name for the sake of completeness, so I would like to thank Aleksey Gurtovoy and David Abrahams for that library, and Daniel Walker for work on those MPL introspection macros.

Finally thanks to Anthony Williams for supplying a workaround for a Visual C++ bug which is needed for introspecting member data where the type of the member data is a compound type.

Index

Reference

Header <boost/tti/gen/has_member_data_gen.hpp>

```
BOOST_TTI_HAS_MEMBER_DATA_GEN(name)
```

Macro BOOST_TTI_HAS_MEMBER_DATA_GEN

BOOST_TTI_HAS_MEMBER_DATA_GEN — Generates the macro metafunction name for BOOST_TTI_HAS_MEMBER_DATA.

Synopsis

```
// In header: <boost/tti/gen/has_member_data_gen.hpp>

BOOST_TTI_HAS_MEMBER_DATA_GEN(name)
```

Description

name = the name of the member data.

returns = the generated macro metafunction name.

Header <boost/tti/gen/has_member_function_gen.hpp>

```
BOOST_TTI_HAS_MEMBER_FUNCTION_GEN(name)
```

Macro BOOST_TTI_HAS_MEMBER_FUNCTION_GEN

BOOST_TTI_HAS_MEMBER_FUNCTION_GEN — Generates the macro metafunction name for BOOST_TTI_HAS_MEMBER_FUNCTION.

Synopsis

```
// In header: <boost/tti/gen/has_member_function_gen.hpp>

BOOST_TTI_HAS_MEMBER_FUNCTION_GEN(name)
```

Description

name = the name of the member function.

returns = the generated macro metafunction name.

Header <boost/tti/gen/has_member_function_with_sig_gen.hpp>

```
BOOST_TTI_HAS_MEMBER_FUNCTION_WITH_SIG_GEN(name)
```

Macro BOOST_TTI_HAS_MEMBER_FUNCTION_WITH_SIG_GEN

BOOST_TTI_HAS_MEMBER_FUNCTION_WITH_SIG_GEN — Generates the macro metafunction name for BOOST_TTI_HAS_MEMBER_FUNCTION_WITH_SIG.

Synopsis

```
// In header: <boost/tti/gen/has_member_function_with_sig_gen.hpp>

BOOST_TTI_HAS_MEMBER_FUNCTION_WITH_SIG_GEN(name)
```

Description

name = the name of the member function.

returns = the generated macro metafunction name.

Header <boost/tti/gen/has_static_member_data_gen.hpp>

```
BOOST_TTI_HAS_STATIC_MEMBER_DATA_GEN(name)
```

Macro BOOST_TTI_HAS_STATIC_MEMBER_DATA_GEN

BOOST_TTI_HAS_STATIC_MEMBER_DATA_GEN — Generates the macro metafunction name for BOOST_TTI_HAS_STATIC_MEMBER_DATA.

Synopsis

```
// In header: <boost/tti/gen/has_static_member_data_gen.hpp>

BOOST_TTI_HAS_STATIC_MEMBER_DATA_GEN(name)
```

Description

name = the name of the static member data.

returns = the generated macro metafunction name.

Header <boost/tti/gen/has_static_member_function_gen.hpp>

```
BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION_GEN(name)
```

Macro BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION_GEN

BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION_GEN — Generates the macro metafunction name for BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION.

Synopsis

```
// In header: <boost/tti/gen/has_static_member_function_gen.hpp>

BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION_GEN(name)
```

Description

name = the name of the static member function.

returns = the generated macro metafunction name.

Header <boost/tti/gen/has_static_member_function_with_sig_gen.hpp>

```


BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION_WITH_SIG_GEN(name)
```

Macro BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION_WITH_SIG_GEN

BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION_WITH_SIG_GEN — Generates the macro metafunction name for BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION_WITH_SIG.

Synopsis

```
// In header: <boost/tti/gen/has_static_member_function_with_sig_gen.hpp>

BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION_WITH_SIG_GEN(name)
```

Description

name = the name of the static member function.

returns = the generated macro metafunction name.

Header <boost/tti/gen/has_template_check_params_gen.hpp>

```


BOOST_TTI_HAS_TEMPLATE_CHECK_PARAMS_GEN(name)
```

Macro BOOST_TTI_HAS_TEMPLATE_CHECK_PARAMS_GEN

BOOST_TTI_HAS_TEMPLATE_CHECK_PARAMS_GEN — Generates the macro metafunction name for BOOST_TTI_HAS_TEMPLATE_CHECK_PARAMS.

Synopsis

```
// In header: <boost/tti/gen/has_template_check_params_gen.hpp>

BOOST_TTI_HAS_TEMPLATE_CHECK_PARAMS_GEN(name)
```

Description

name = the name of the class template.

returns = the generated macro metafunction name.

Header <boost/tti/gen/has_template_gen.hpp>

```


BOOST_TTI_HAS_TEMPLATE_GEN(name)
```

Macro BOOST_TTI_HAS_TEMPLATE_GEN

BOOST_TTI_HAS_TEMPLATE_GEN — Generates the macro metafunction name for BOOST_TTI_HAS_TEMPLATE.

Synopsis

```
// In header: <boost/tti/gen/has_template_gen.hpp>

BOOST_TTI_HAS_TEMPLATE_GEN(name)
```

Description

name = the name of the class template.

returns = the generated macro metafunction name.

Header <boost/tti/gen/has_type_gen.hpp>

```


BOOST_TTI_HAS_TYPE_GEN(name)
```

Macro BOOST_TTI_HAS_TYPE_GEN

BOOST_TTI_HAS_TYPE_GEN — Generates the macro metafunction name for BOOST_TTI_HAS_TYPE.

Synopsis

```
// In header: <boost/tti/gen/has_type_gen.hpp>

BOOST_TTI_HAS_TYPE_GEN(name)
```

Description

name = the name of the type.

returns = the generated macro metafunction name.

Header <boost/tti/gen/member_type_gen.hpp>

```
BOOST_TTI_MEMBER_TYPE_GEN(name)
```

Macro BOOST_TTI_MEMBER_TYPE_GEN

BOOST_TTI_MEMBER_TYPE_GEN — Generates the macro metafunction name for BOOST_TTI_MEMBER_TYPE.

Synopsis

```
// In header: <boost/tti/gen/member_type_gen.hpp>

BOOST_TTI_MEMBER_TYPE_GEN(name)
```

Description

name = the name of the inner type.

returns = the generated macro metafunction name.

Header <boost/tti/gen/namespace_gen.hpp>

```
BOOST_TTI_NAMESPACE
```

Macro BOOST_TTI_NAMESPACE

BOOST_TTI_NAMESPACE — Generates the name of the Boost TTI namespace.

Synopsis

```
// In header: <boost/tti/gen/namespace_gen.hpp>

BOOST_TTI_NAMESPACE
```

Description

returns = the generated name of the Boost TTI namespace.

Header <boost/tti/gen/vm_has_template_check_params_gen.hpp>

```
BOOST_TTI_VM_HAS_TEMPLATE_CHECK_PARAMS_GEN(name)
```

Macro BOOST_TTI_VM_HAS_TEMPLATE_CHECK_PARAMS_GEN

BOOST_TTI_VM_HAS_TEMPLATE_CHECK_PARAMS_GEN — Generates the macro metafunction name for BOOST_TTI_VM_HAS_TEMPLATE_CHECK_PARAMS.

Synopsis

```
// In header: <boost/tti/gen/vm_has_template_check_params_gen.hpp>

BOOST_TTI_VM_HAS_TEMPLATE_CHECK_PARAMS_GEN(name)
```

Description

name = the name of the class template.

returns = the generated macro metafunction name.

Header <boost/tti/has_member_data.hpp>

```
BOOST_TTI_TRAIT_HAS_MEMBER_DATA(trait, name)
BOOST_TTI_HAS_MEMBER_DATA(name)
```

Macro BOOST_TTI_TRAIT_HAS_MEMBER_DATA

BOOST_TTI_TRAIT_HAS_MEMBER_DATA — Expands to a metafunction which tests whether a member data with a particular name and type exists.

Synopsis

```
// In header: <boost/tti/has_member_data.hpp>

BOOST_TTI_TRAIT_HAS_MEMBER_DATA(trait, name)
```

Description

trait = the name of the metafunction within the tti namespace.

name = the name of the inner member.

returns = a metafunction called "boost::tti::trait" where 'trait' is the macro parameter.

The metafunction types and return:

TTI_T = the enclosing type in which to look for our 'name'.

TTI_R = the type of the member data.

returns = 'value' is true if the 'name' exists, with the appropriate type, otherwise 'value' is false.

Macro BOOST_TTI_HAS_MEMBER_DATA

BOOST_TTI_HAS_MEMBER_DATA — Expands to a metafunction which tests whether a member data with a particular name and type exists.

Synopsis

```
// In header: <boost/tti/has_member_data.hpp>

BOOST_TTI_HAS_MEMBER_DATA(name)
```

Description

name = the name of the inner member.

returns = a metafunction called "boost::tti::has_member_data_name" where 'name' is the macro parameter.

The metafunction types and return:

TTI_T = the enclosing type in which to look for our 'name'.

TTI_R = the type of the member data.

returns = 'value' is true if the 'name' exists, with the appropriate type, otherwise 'value' is false.

Header <boost/tti/has_member_function.hpp>

```


BOOST_TTI_TRAIT_HAS_MEMBER_FUNCTION(trait, name)
BOOST_TTI_HAS_MEMBER_FUNCTION(name)
```

Macro BOOST_TTI_TRAIT_HAS_MEMBER_FUNCTION

BOOST_TTI_TRAIT_HAS_MEMBER_FUNCTION — Expands to a metafunction which tests whether a member function with a particular name and signature exists.

Synopsis

```
// In header: <boost/tti/has_member_function.hpp>

BOOST_TTI_TRAIT_HAS_MEMBER_FUNCTION(trait, name)
```

Description

trait = the name of the metafunction within the tti namespace.

name = the name of the inner member.

returns = a metafunction called "boost::tti::trait" where 'trait' is the macro parameter.

The metafunction types and return:

TTI_T = the enclosing type in which to look for our 'name'.

TTI_R = the return type of the member function.

TTI_FS = an optional parameter which are the parameters of the member function as a boost::mpl forward sequence.

TTI_TAG = an optional parameter which is a boost::function_types tag to apply to the member function.

returns = 'value' is true if the 'name' exists, with the appropriate type, otherwise 'value' is false.

Macro BOOST_TTI_HAS_MEMBER_FUNCTION

BOOST_TTI_HAS_MEMBER_FUNCTION — Expands to a metafunction which tests whether a member function with a particular name and signature exists.

Synopsis

```
// In header: <boost/tti/has_member_function.hpp>

BOOST_TTI_HAS_MEMBER_FUNCTION(name)
```

Description

name = the name of the inner member.

returns = a metafunction called "boost::tti::has_member_function_name" where 'name' is the macro parameter.

The metafunction types and return:

TTI_T = the enclosing type in which to look for our 'name'.

TTI_R = the return type of the member function.

TTI_FS = an optional parameter which are the parameters of the member function as a boost::mpl forward sequence.

TTI_TAG = an optional parameter which is a boost::function_types tag to apply to the member function.

returns = 'value' is true if the 'name' exists, with the appropriate type, otherwise 'value' is false.

Header <boost/tti/has_member_function_with_sig.hpp>

```


BOOST_TTI_TRAIT_HAS_MEMBER_FUNCTION_WITH_SIG(trait, name)
BOOST_TTI_HAS_MEMBER_FUNCTION_WITH_SIG(name)
```

Macro BOOST_TTI_TRAIT_HAS_MEMBER_FUNCTION_WITH_SIG

BOOST_TTI_TRAIT_HAS_MEMBER_FUNCTION_WITH_SIG — Expands to a metafunction which tests whether a member function with a particular name and composite type exists.

Synopsis

```
// In header: <boost/tti/has_member_function_with_sig.hpp>

BOOST_TTI_TRAIT_HAS_MEMBER_FUNCTION_WITH_SIG(trait, name)
```

Description

trait = the name of the metafunction within the tti namespace.

name = the name of the inner member.

returns = a metafunction called "boost::tti::trait" where 'trait' is the macro parameter.

The metafunction types and return:

TTI_T = the member function type, in the form of a member function pointer - 'return_type (Class::*)(parameter_types...)', in which to look for our 'name'.

returns = 'value' is true if the 'name' exists, with the appropriate type, otherwise 'value' is false.

Macro BOOST_TTI_HAS_MEMBER_FUNCTION_WITH_SIG

BOOST_TTI_HAS_MEMBER_FUNCTION_WITH_SIG — Expands to a metafunction which tests whether a member function with a particular name and composite type exists.

Synopsis

```
// In header: <boost/tti/has_member_function_with_sig.hpp>

BOOST_TTI_HAS_MEMBER_FUNCTION_WITH_SIG(name)
```

Description

name = the name of the inner member.

returns = a metafunction called "boost::tti::has_comp_member_function_name" where 'name' is the macro parameter.

The metafunction types and return:

TTI_T = the member function type, in the form of a member function pointer - 'return_type (Class::*)(parameter_types...)', in which to look for our 'name'.

returns = 'value' is true if the 'name' exists, with the appropriate type, otherwise 'value' is false.

Header <boost/tti/has_static_member_data.hpp>

```


BOOST_TTI_TRAIT_HAS_STATIC_MEMBER_DATA(trait, name)
BOOST_TTI_HAS_STATIC_MEMBER_DATA(name)
```

Macro BOOST_TTI_TRAIT_HAS_STATIC_MEMBER_DATA

BOOST_TTI_TRAIT_HAS_STATIC_MEMBER_DATA — Expands to a metafunction which tests whether a static member data with a particular name and type exists.

Synopsis

```
// In header: <boost/tti/has_static_member_data.hpp>

BOOST_TTI_TRAIT_HAS_STATIC_MEMBER_DATA(trait, name)
```

Description

trait = the name of the metafunction within the tti namespace.

name = the name of the inner member.

returns = a metafunction called "boost::tti::trait" where 'trait' is the macro parameter.

The metafunction types and return:

TTI_T = the enclosing type.

TTI_Type = the static member data type, in the form of a data type, in which to look for our 'name'.

returns = 'value' is true if the 'name' exists within the enclosing type, with the appropriate type, otherwise 'value' is false.

Macro BOOST_TTI_HAS_STATIC_MEMBER_DATA

BOOST_TTI_HAS_STATIC_MEMBER_DATA — Expands to a metafunction which tests whether a static member data with a particular name and type exists.

Synopsis

```
// In header: <boost/tti/has_static_member_data.hpp>

BOOST_TTI_HAS_STATIC_MEMBER_DATA(name)
```

Description

name = the name of the inner member.

returns = a metafunction called "boost::tti::has_static_member_name" where 'name' is the macro parameter.

The metafunction types and return:

TTI_T = the enclosing type.

TTI_Type = the static member data type, in the form of a data type, in which to look for our 'name'.

returns = 'value' is true if the 'name' exists within the enclosing type, with the appropriate type, otherwise 'value' is false.

Header <boost/tti/has_static_member_function.hpp>

```
BOOST_TTI_TRAIT_HAS_STATIC_MEMBER_FUNCTION(trait, name)
BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION(name)
```

Macro BOOST_TTI_TRAIT_HAS_STATIC_MEMBER_FUNCTION

BOOST_TTI_TRAIT_HAS_STATIC_MEMBER_FUNCTION — Expands to a metafunction which tests whether a static member function with a particular name and signature exists.

Synopsis

```
// In header: <boost/tti/has_static_member_function.hpp>

BOOST_TTI_TRAIT_HAS_STATIC_MEMBER_FUNCTION(trait, name)
```

Description

trait = the name of the metafunction within the tti namespace.

name = the name of the inner member.

returns = a metafunction called "boost::tti::trait" where 'trait' is the macro parameter.

The metafunction types and return:

TTI_T = the enclosing type in which to look for our 'name'.

TTI_R = the return type of the static member function.

TTI_FS = an optional parameter which are the parameters of the static member function as a boost::mpl forward sequence.

TTI_TAG = an optional parameter which is a boost::function_types tag to apply to the static member function.

returns = 'value' is true if the 'name' exists, with the appropriate type, otherwise 'value' is false.

Macro BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION

BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION — Expands to a metafunction which tests whether a static member function with a particular name and signature exists.

Synopsis

```
// In header: <boost/tti/has_static_member_function.hpp>

BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION(name)
```

Description

name = the name of the inner member.

returns = a metafunction called "boost::tti::has_static_member_function_name" where 'name' is the macro parameter.

The metafunction types and return:

TTI_T = the enclosing type in which to look for our 'name'.

TTI_R = the return type of the static member function.

TTI_FS = an optional parameter which are the parameters of the static member function as a boost::mpl forward sequence.

TTI_TAG = an optional parameter which is a boost::function_types tag to apply to the static member function.

returns = 'value' is true if the 'name' exists, with the appropriate type, otherwise 'value' is false.

Header <boost/tti/has_static_member_function_with_sig.hpp>

```


BOOST_TTI_TRAIT_HAS_STATIC_MEMBER_FUNCTION_WITH_SIG(trait, name)
BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION_WITH_SIG(name)
```

Macro BOOST_TTI_TRAIT_HAS_STATIC_MEMBER_FUNCTION_WITH_SIG

BOOST_TTI_TRAIT_HAS_STATIC_MEMBER_FUNCTION_WITH_SIG — Expands to a metafunction which tests whether a static member function with a particular name and composite type exists.

Synopsis

```
// In header: <boost/tti/has_static_member_function_with_sig.hpp>

BOOST_TTI_TRAIT_HAS_STATIC_MEMBER_FUNCTION_WITH_SIG(trait, name)
```

Description

trait = the name of the metafunction within the tti namespace.

name = the name of the inner member.

returns = a metafunction called "boost::tti::trait" where 'trait' is the macro parameter.

The metafunction types and return:

TTI_T = the enclosing type.

TTI_Type = the static member function type, in the form of a composite function type - 'return_type (parameter_types...)', in which to look for our 'name'.

returns = 'value' is true if the 'name' exists within the enclosing type, with the appropriate type, otherwise 'value' is false.

Macro BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION_WITH_SIG

BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION_WITH_SIG — Expands to a metafunction which tests whether a static member function with a particular name and composite type exists.

Synopsis

```
// In header: <boost/tti/has_static_member_function_with_sig.hpp>

BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION_WITH_SIG(name)
```

Description

name = the name of the inner member.

returns = a metafunction called "boost::tti::has_comp_static_member_function_name" where 'name' is the macro parameter.

The metafunction types and return:

TTI_T = the enclosing type.

TTI_Type = the static member function type, in the form of a composite function type - 'return_type (parameter_types...)', in which to look for our 'name'.

returns = 'value' is true if the 'name' exists within the enclosing type, with the appropriate type, otherwise 'value' is false.

Header <boost/tti/has_template.hpp>

```


BOOST_TTI_TRAIT_HAS_TEMPLATE(trait, name)
BOOST_TTI_HAS_TEMPLATE(name)
```

Macro BOOST_TTI_TRAIT_HAS_TEMPLATE

BOOST_TTI_TRAIT_HAS_TEMPLATE — Expands to a metafunction which tests whether an inner class template with a particular name exists.

Synopsis

```
// In header: <boost/tti/has_template.hpp>

BOOST_TTI_TRAIT_HAS_TEMPLATE(trait, name)
```

Description

trait = the name of the metafunction within the tti namespace.

name = the name of the inner template.

returns = a metafunction called "boost::tti::trait" where 'trait' is the macro parameter.

The metafunction types and return:

TTI_T = the enclosing type in which to look for our 'name'.

returns = 'value' is true if the 'name' template exists within the enclosing type, otherwise 'value' is false.

The template must have all 'class' (or 'typename') parameters types.

Macro BOOST_TTI_HAS_TEMPLATE

BOOST_TTI_HAS_TEMPLATE — Expands to a metafunction which tests whether an inner class template with a particular name exists.

Synopsis

```
// In header: <boost/tti/has_template.hpp>

BOOST_TTI_HAS_TEMPLATE(name)
```

Description

name = the name of the inner template.

returns = a metafunction called "boost::tti::has_template_name" where 'name' is the macro parameter.

The metafunction types and return:

TTI_T = the enclosing type in which to look for our 'name'.

returns = 'value' is true if the 'name' template exists within the enclosing type, otherwise 'value' is false.

The template must have all 'class' (or 'typename') parameters types.

Header <boost/tti/has_template_check_params.hpp>

```


BOOST_TTI_TRAIT_HAS_TEMPLATE_CHECK_PARAMS(trait, name, tpSeq)
BOOST_TTI_HAS_TEMPLATE_CHECK_PARAMS(name, tpSeq)
```

Macro BOOST_TTI_TRAIT_HAS_TEMPLATE_CHECK_PARAMS

BOOST_TTI_TRAIT_HAS_TEMPLATE_CHECK_PARAMS — Expands to a metafunction which tests whether an inner class template with a particular name and signature exists.

Synopsis

```
// In header: <boost/tti/has_template_check_params.hpp>

BOOST_TTI_TRAIT_HAS_TEMPLATE_CHECK_PARAMS(trait, name, tpSeq)
```

Description

trait = the name of the metafunction within the tti namespace.

name = the name of the inner class template.

tpSeq = a Boost PP sequence which has the class template parameters. Each part of the template parameters separated by a comma (,) is put in a separate sequence element.

returns = a metafunction called "boost::tti::trait" where 'trait' is the macro parameter.

The metafunction types and return:

TTI_T = the enclosing type in which to look for our 'name'.

returns = 'value' is true if the 'name' class template with the signature as defined by the 'tpSeq' exists within the enclosing type, otherwise 'value' is false.

Macro BOOST_TTI_HAS_TEMPLATE_CHECK_PARAMS

BOOST_TTI_HAS_TEMPLATE_CHECK_PARAMS — Expands to a metafunction which tests whether an inner class template with a particular name and signature exists.

Synopsis

```
// In header: <boost/tti/has_template_check_params.hpp>

BOOST_TTI_HAS_TEMPLATE_CHECK_PARAMS(name, tpSeq)
```

Description

name = the name of the inner class template.

tpSeq = a Boost PP sequence which has the class template parameters. Each part of the template parameters separated by a comma (,) is put in a separate sequence element.

returns = a metafunction called "boost::tti::has_template_check_params_name" where 'name' is the macro parameter.

The metafunction types and return:

TTI_T = the enclosing type in which to look for our 'name'.

returns = 'value' is true if the 'name' class template with the signature as defined by the 'tpSeq' exists within the enclosing type, otherwise 'value' is false.

Header <boost/tti/has_type.hpp>

```


BOOST_TTI_TRAIT_HAS_TYPE(trait, name)
BOOST_TTI_HAS_TYPE(name)
```

Macro BOOST_TTI_TRAIT_HAS_TYPE

BOOST_TTI_TRAIT_HAS_TYPE — Expands to a metafunction which tests whether an inner type with a particular name exists and optionally is a particular type.

Synopsis

```
// In header: <boost/tti/has_type.hpp>

BOOST_TTI_TRAIT_HAS_TYPE(trait, name)
```

Description

trait = the name of the metafunction within the tti namespace.

name = the name of the inner type.

returns = a metafunction called "boost::tti::trait" where 'trait' is the macro parameter.

The metafunction types and return:

TTI_T = the enclosing type in which to look for our 'name'.

TTI_U = the type of the inner type named 'name' as an optional parameter.

returns = 'value' is true if the 'name' type exists within the enclosing type and, if type U is specified, the 'name' type is the same as the type U, otherwise 'value' is false.

Macro BOOST_TTI_HAS_TYPE

BOOST_TTI_HAS_TYPE — Expands to a metafunction which tests whether an inner type with a particular name exists and optionally is a particular type.

Synopsis

```
// In header: <boost/tti/has_type.hpp>

BOOST_TTI_HAS_TYPE(name)
```

Description

name = the name of the inner type.

returns = a metafunction called "boost::tti::has_type_name" where 'name' is the macro parameter.

The metafunction types and return:

TTI_T = the enclosing type in which to look for our 'name'.

TTI_U = the type of the inner type named 'name' as an optional parameter.

returns = 'value' is true if the 'name' type exists within the enclosing type and, if type U is specified, the 'name' type is the same as the type U, otherwise 'value' is false.

Header <boost/tti/member_type.hpp>

```


BOOST_TTI_TRAIT_MEMBER_TYPE(trait, name)
BOOST_TTI_MEMBER_TYPE(name)
```

Macro BOOST_TTI_TRAIT_MEMBER_TYPE

BOOST_TTI_TRAIT_MEMBER_TYPE — Expands to a metafunction whose typedef 'type' is either the named type or an unspecified type.

Synopsis

```
// In header: <boost/tti/member_type.hpp>

BOOST_TTI_TRAIT_MEMBER_TYPE(trait, name)
```

Description

trait = the name of the metafunction within the tti namespace.

name = the name of the inner type.

returns = a metafunction called "boost::tti::trait" where 'trait' is the macro parameter.

The metafunction types and return:

TTI_T = the enclosing type.

returns = 'type' is the inner type of 'name' if the inner type exists within the enclosing type, else 'type' is an unspecified type.

The purpose of this macro is to encapsulate the 'name' type as the typedef 'type' of a metafunction, but only if it exists within the enclosing type. This allows for a lazy evaluation of inner type existence which can be used by other metafunctions in this library.

Macro BOOST_TTI_MEMBER_TYPE

BOOST_TTI_MEMBER_TYPE — Expands to a metafunction whose typedef 'type' is either the named type or an unspecified type.

Synopsis

```
// In header: <boost/tti/member_type.hpp>

BOOST_TTI_MEMBER_TYPE(name)
```

Description

name = the name of the inner type.

returns = a metafunction called "boost::tti::member_type_name" where 'name' is the macro parameter.

The metafunction types and return:

TTI_T = the enclosing type.

returns = 'type' is the inner type of 'name' if the inner type exists within the enclosing type, else 'type' is an unspecified type.

The purpose of this macro is to encapsulate the 'name' type as the typedef 'type' of a metafunction, but only if it exists within the enclosing type. This allows for a lazy evaluation of inner type existence which can be used by other metafunctions in this library.

Header <boost/tti/mf/mf_has_member_data.hpp>

```
namespace boost {
  namespace tti {
    template<typename TTI_HasMemberData, typename TTI_T, typename TTI_R> 
      struct mf_has_member_data;
  }
}
```

Struct template mf_has_member_data

boost::tti::mf_has_member_data — A metafunction which checks whether a member data exists within an enclosing type.

Synopsis

```
// In header: <boost/tti/mf/mf_has_member_data.hpp>

template<typename TTI_HasMemberData, typename TTI_T, typename TTI_R>
struct mf_has_member_data {
};
```

Description

This metafunction takes its specific types as nullary metafunctions whose typedef 'type' member is the actual type used.

The metafunction types and return:

TTI_HasMemberData = a Boost MPL lambda expression using the metafunction generated from the TTI_HAS_MEMBER_DATA (or TTI_TRAIT_HAS_MEMBER_DATA) macro.
The easiest way to generate the lambda expression is to use a Boost MPL placeholder expression of the form 'metafunction<_,_>'. You can also use a metafunction class generated by boost::mpl::quote2.

TTI_T = the enclosing type as a nullary metafunction.

TTI_R = the type of the member data as a nullary metafunction.

returns = 'value' is true if the member data exists within the enclosing type, otherwise 'value' is false.

Header <boost/tti/mf/mf_has_member_function.hpp>

```
namespace boost {
  namespace tti {
    template<typename TTI_HasMemberFunction, typename TTI_T, typename TTI_R,
             typename TTI_FS = boost::mpl::vector<>,
             typename TTI_TAG = boost::function_types::null_tag>
      struct mf_has_member_function;
  }
}
```

Struct template mf_has_member_function

boost::tti::mf_has_member_function — A metafunction which checks whether a member function exists within an enclosing type.

Synopsis

```
// In header: <boost/tti/mf/mf_has_member_function.hpp>

template<typename TTI_HasMemberFunction, typename TTI_T, typename TTI_R,
         typename TTI_FS = boost::mpl::vector<>,
         typename TTI_TAG = boost::function_types::null_tag>
struct mf_has_member_function {
};
```

Description

This metafunction takes its specific types, except for the optional parameters, as nullary metafunctions whose typedef 'type' member is the actual type used.

The metafunction types and return:

TTI_HasMemberFunction = a Boost MPL lambda expression using the metafunction generated from the TTI_HAS_MEMBER_FUNCTION (or TTI_TRAIT_HAS_MEMBER_FUNCTION) macro.
The easiest way to generate the lambda expression is to use a Boost MPL placeholder expression of the form 'metafunction<_,_> (or optionally 'metafunction<_,_,_> or ' 'metafunction<_,_,_,_>)'. You can also use a metafunction class generated by boost::mpl::quote4.

TTI_T = the enclosing type as a nullary metafunction.

TTI_R = the return type of the member function as a nullary metafunction.

TTI_FS = an optional parameter which is the parameters of the member function, each as a nullary metafunction, as a boost::mpl forward sequence.
This parameter defaults to boost::mpl::vector<>.

TTI_TAG = an optional parameter which is a boost::function_types tag to apply to the member function.
This parameter defaults to boost::function_types::null_tag.

returns = 'value' is true if the member function exists within the enclosing type, otherwise 'value' is false.

Header <boost/tti/mf/mf_has_static_member_data.hpp>

```
namespace boost {
  namespace tti {
    template<typename TTI_HasStaticMemberData, typename TTI_T, typename TTI_R>
      struct mf_has_static_member_data;
  }
}
```

Struct template mf_has_static_member_data

boost::tti::mf_has_static_member_data — A metafunction which checks whether a static member data exists within an enclosing type.

Synopsis

```
// In header: <boost/tti/mf/mf_has_static_member_data.hpp>

template<typename TTI_HasStaticMemberData, typename TTI_T, typename TTI_R>
struct mf_has_static_member_data {
};
```

Description

This metafunction takes its specific types as nullary metafunctions whose typedef 'type' member is the actual type used.

The metafunction types and return:

TTI_HasStaticMemberData = a Boost MPL lambda expression using the metafunction generated from the TTI_HAS_STATIC_MEMBER_DATA (or TTI_TRAIT_HAS_STATIC_MEMBER_DATA) macro.
The easiest way to generate the lambda expression is to use a Boost MPL placeholder expression of the form 'metafunction<_,_>'. You can also use a metafunction class generated by boost::mpl::quote2.

TTI_T = the enclosing type as a nullary metafunction.

TTI_R = the type of the static member data as a nullary metafunction.

returns = 'value' is true if the member data exists within the enclosing type, otherwise 'value' is false.

Header <boost/tti/mf/mf_has_static_member_function.hpp>

```
namespace boost {
  namespace tti {
    template<typename TTI_HasStaticMemberFunction, typename TTI_T,
             typename TTI_R, typename TTI_FS = boost::mpl::vector<>,
             typename TTI_TAG = boost::function_types::null_tag>
      struct mf_has_static_member_function;
  }
}
```

Struct template mf_has_static_member_function

boost::tti::mf_has_static_member_function — A metafunction which checks whether a static member function exists within an enclosing type.

Synopsis

```
// In header: <boost/tti/mf/mf_has_static_member_function.hpp>

template<typename TTI_HasStaticMemberFunction, typename TTI_T, typename TTI_R,
         typename TTI_FS = boost::mpl::vector<>,
         typename TTI_TAG = boost::function_types::null_tag>
struct mf_has_static_member_function {
};
```

Description

This metafunction takes its specific types, except for the optional parameters, as nullary metafunctions whose typedef 'type' member is the actual type used.

The metafunction types and return:

TTI_HasStaticMemberFunction = a Boost MPL lambda expression using the metafunction generated from the TTI_HAS_STATIC_MEMBER_FUNCTION (or TTI_TRAIT_HAS_STATIC_MEMBER_FUNCTION) macro.
The easiest way to generate the lambda expression is to use a Boost MPL placeholder expression of the form 'metafunction<_,_> (or optionally 'metafunction<_,_,_> or ' 'metafunction<_,_,_,_>)'. You can also use a metafunction class generated by boost::mpl::quote4.

TTI_T = the enclosing type as a nullary metafunction.

TTI_R = the return type of the static member function as a nullary metafunction.

TTI_FS = an optional parameter which is the parameters of the static member function, each as a nullary metafunction, as a boost::mpl forward sequence.

TTI_TAG = an optional parameter which is a boost::function_types tag to apply to the static member function.

returns = 'value' is true if the member function exists within the enclosing type, otherwise 'value' is false.

Header <boost/tti/mf/mf_has_template.hpp>

```
namespace boost {
  namespace tti {
    template<typename TTI_HasTemplate, typename TTI_T> struct mf_has_template;
  }
}
```

Struct template mf_has_template

boost::tti::mf_has_template — A metafunction which checks whether a class template exists within an enclosing type.

Synopsis

```
// In header: <boost/tti/mf/mf_has_template.hpp>

template<typename TTI_HasTemplate, typename TTI_T>
struct mf_has_template {
};
```

Description

This metafunction takes its enclosing type as nullary metafunctions whose typedef 'type' member is the actual type used.

The metafunction types and return:

TTI_HasTemplate = a Boost MPL lambda expression using the metafunction generated from the TTI_HAS_TEMPLATE (TTI_TRAIT_HAS_TEMPLATE) macro.
The easiest way to generate the lambda expression is to use a Boost MPL placeholder expression of the form 'metafunction<_>'. You can also use a metafunction class generated by boost::mpl::quote1.

TTI_T = the enclosing type as a nullary metafunction.

returns = 'value' is true if the template exists within the enclosing type, otherwise 'value' is false.

Header <boost/tti/mf/mf_has_template_check_params.hpp>

```
namespace boost {
  namespace tti {
    template<typename TTI_HasTemplateCheckParams, typename TTI_T>
      struct mf_has_template_check_params;
  }
}
```

Struct template mf_has_template_check_params

boost::tti::mf_has_template_check_params — A metafunction which checks whether a class template with its parameters exists within an enclosing type.

Synopsis

```
// In header: <boost/tti/mf/mf_has_template_check_params.hpp>

template<typename TTI_HasTemplateCheckParams, typename TTI_T>
struct mf_has_template_check_params {
};
```

Description

This metafunction takes its enclosing type as nullary metafunctions whose typedef 'type' member is the actual type used.

The metafunction types and return:

TTI_HasTemplateCheckParams = a Boost MPL lambda expression using the metafunction generated from either the TTI_HAS_TEMPLATE_CHECK_PARAMS (TTI_TRAIT_HAS_TEMPLATE_CHECK_PARAMS) or TTI_VM_HAS_TEMPLATE_CHECK_PARAMS (TTI_VM_TRAIT_HAS_TEMPLATE_CHECK_PARAMS) macros.
The easiest way to generate the lambda expression is to use a Boost MPL placeholder expression of the form 'metafunction<_>'. You can also use a metafunction class generated by boost::mpl::quote1.

TTI_T = The enclosing type as a nullary metafunction.

returns = 'value' is true if the template exists within the enclosing type, otherwise 'value' is false.

Header <boost/tti/mf/mf_has_type.hpp>

```
namespace boost {
  namespace tti {
    template<typename TTI_HasType, typename TTI_T, 
             typename TTI_U = boost::mpl::identity<BOOST_TTI_NAMESPACE::detail::notype> > 
      struct mf_has_type;
  }
}
```

Struct template mf_has_type

boost::tti::mf_has_type — A metafunction which checks whether a type exists within an enclosing type and optionally is a particular type.

Synopsis

```
// In header: <boost/tti/mf/mf_has_type.hpp>

template<typename TTI_HasType, typename TTI_T, 
         typename TTI_U = boost::mpl::identity<BOOST_TTI_NAMESPACE::detail::notype> > 
struct mf_has_type {
};
```

Description

This metafunction takes its specific types as nullary metafunctions whose typedef 'type' member is the actual type used.

The metafunction types and return:

TTI_HasType = a Boost MPL lambda expression using the metafunction generated from the TTI_HAS_TYPE (or TTI_TRAIT_HAS_TYPE) macro.

The easiest way to generate the lambda expression is to use a Boost MPL placeholder expression of the form 'metafunction<_>' (or optionally 'metafunction<_,_>'). You can also use a metafunction class generated by boost::mpl::quote2.

TTI_T = the enclosing type as a nullary metafunction.

TTI_U = the type of the inner type as a nullary metafunction, as an optional parameter.

returns = 'value' is true if the type exists within the enclosing type and, if type U is specified, the type is the same as the type U, otherwise 'value' is false.

Header <boost/tti/mf/mf_member_type.hpp>

```
namespace boost {
  namespace tti {
    template<typename TTI_T> struct valid_member_type;
    template<typename TTI_T> struct mf_valid_member_type;
    template<typename TTI_MemberType, typename TTI_T> struct mf_member_type;
  }
}
```

Struct template valid_member_type

boost::tti::valid_member_type — A metafunction which checks whether the member 'type' returned from invoking the macro metafunction generated by TTI_MEMBER_TYPE (TTI_TRAIT_MEMBER_TYPE) or from invoking boost::tti::mf_member_type is a valid type.

Synopsis

```
// In header: <boost/tti/mf/mf_member_type.hpp>

template<typename TTI_T>
struct valid_member_type {
};
```

Description

The metafunction types and return:

TTI_T = returned inner 'type' from invoking the macro metafunction generated by TTI_MEMBER_TYPE (TTI_TRAIT_MEMBER_TYPE) or from invoking boost::tti::mf_member_type.

returns = 'value' is true if the type is valid, otherwise 'value' is false.

Struct template mf_valid_member_type

boost::tti::mf_valid_member_type — A metafunction which checks whether the member 'type' returned from invoking the macro metafunction generated by TTI_MEMBER_TYPE (TTI_TRAIT_MEMBER_TYPE) or from invoking boost::tti::mf_member_type is a valid type.

Synopsis

```
// In header: <boost/tti/mf/mf_member_type.hpp>

template<typename TTI_T>
struct mf_valid_member_type {
};
```

Description

The metafunction types and return:

TTI_T = the nullary metafunction from invoking the macro metafunction generated by TTI_MEMBER_TYPE (TTI_TRAIT_MEMBER_TYPE) or from invoking boost::tti::mf_member_type.

returns = 'value' is true if the type is valid, otherwise 'value' is false.

Struct template mf_member_type

boost::tti::mf_member_type — A metafunction whose typedef 'type' is either the internal type or an unspecified type.

Synopsis

```
// In header: <boost/tti/mf/mf_member_type.hpp>

template<typename TTI_MemberType, typename TTI_T>
struct mf_member_type {
};
```

Description

This metafunction takes its enclosing type as a nullary metafunction whose typedef 'type' member is the actual type used.

The metafunction types and return:

TTI_MemberType = a Boost MPL lambda expression using the metafunction generated from the TTI_MEMBER_TYPE (or TTI_TRAIT_MEMBER_TYPE) macro.
The easiest way to generate the lambda expression is to use a Boost MPL placeholder expression of the form 'metafunction<_>'. You can also use a metafunction class generated by boost::mpl::quote1.

TTI_T = the enclosing type as a nullary metafunction.

returns = 'type' is the inner type of the 'name' in TTI_MEMBER_TYPE (or TTI_TRAIT_MEMBER_TYPE) if the inner type exists within the enclosing type, else 'type' is an unspecified type.

The purpose of this metafunction is to encapsulate the 'name' type in TTI_MEMBER_TYPE (or TTI_TRAIT_MEMBER_TYPE) as the typedef 'type' of a metafunction, but only if it exists within the enclosing type. This allows for a lazy evaluation of inner type existence which can be used by other metafunctions in this library.

Furthermore this metafunction allows the enclosing type to be return type from either the metafunction generated from TTI_MEMBER_TYPE (or TTI_TRAIT_MEMBER_TYPE) or from this metafunction itself.

Header <boost/tti/vm_has_template_check_params.hpp>

```
BOOST_TTI_VM_TRAIT_HAS_TEMPLATE_CHECK_PARAMS(trait, name, ...)
BOOST_TTI_VM_HAS_TEMPLATE_CHECK_PARAMS(name, ...)
```

Macro BOOST_TTI_VM_TRAIT_HAS_TEMPLATE_CHECK_PARAMS

BOOST_TTI_VM_TRAIT_HAS_TEMPLATE_CHECK_PARAMS — Expands to a metafunction which tests whether an inner class template with a particular name and signature exists.

Synopsis

```
// In header: <boost/tti/vm_has_template_check_params.hpp>

BOOST_TTI_VM_TRAIT_HAS_TEMPLATE_CHECK_PARAMS(trait, name, ...)
```

Description

trait = the name of the metafunction within the tti namespace.

name = the name of the inner class template.

... = variadic macro data which has the class template parameters.

returns = a metafunction called "boost::tti::trait" where 'trait' is the macro parameter.

The metafunction types and return:

TTI_T = the enclosing type in which to look for our 'name'.

returns = 'value' is true if the 'name' class template, with the signature as defined by the '...' variadic macro data, exists within the enclosing type, otherwise 'value' is false.

Macro BOOST_TTI_VM_HAS_TEMPLATE_CHECK_PARAMS

BOOST_TTI_VM_HAS_TEMPLATE_CHECK_PARAMS — Expands to a metafunction which tests whether an inner class template with a particular name and signature exists.

Synopsis

```
// In header: <boost/tti/vm_has_template_check_params.hpp>

BOOST_TTI_VM_HAS_TEMPLATE_CHECK_PARAMS(name, ...)
```

Description

name = the name of the inner class template.

... = variadic macro data which has the class template parameters.

returns = a metafunction called "boost::tti::has_template_check_params_name" where 'name' is the macro parameter.

The metafunction types and return:

TTI_T = the enclosing type in which to look for our 'name'.

returns = 'value' is true if the 'name' class template, with the signature as defined by the '...' variadic macro data, exists within the enclosing type, otherwise 'value' is false.