The Type Traits Introspection Library

Edward Diener

Table of Contents

Introduction

Welcome to the Boost Type Traits Introspection library, abbreviated TTI.

TTI is a library which provides the ability to introspect by name the elements of a type at compile time.

TTI works through macros generating metafunctions. Metafunctions are class templates of a particular syntax, having a nested 'type' member. So wherever in C++ class templates can occur, TTI macros can be used. The metafunctions generated by TTI are no different from any other metafunction as defined by the Boost MPL library.

The metafunctions generated by TTI are used to introspect elements of a type at compile time, always passing at minimum to each metafunction the enclosing type being introspected.

The name of the library has been 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 within that type.

I use the word "introspect" in a very broad sense here. Normally computer 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 know and supply the name, and use the functionality provided for the correct type of inner element to find out if that 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 the means to create a typedef for a type which may not exist. This typedef type can be used as a type in the metafunctions of the library without producing compile-time errors.

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 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 a library in order to use the TTI library.

Header Files

There are is a single header file, `boost/tti/tti.hpp`, which includes all the header files in the library.

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 the nested elements to be introspected. These header files are:

Table 1. TTI Header Files

Introspected Element	Specific Header File
Type	`has_type.hpp`
Class Template	`has_template.hpp`
Member data	`has_member_data.hpp`
Member function	`has_member_function.hpp`
Static member data	`has_static_member_data.hpp`
Static member function	`has_static_member_function.hpp`
Data	`has_data.hpp`
Function	`has_function.hpp`
Member Type Creation	`member_type.hpp`

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 expanded.

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.

General Functionality

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

- Does it have a nested type with a particular name ?
- Does it have a nested type with a particular name which fulfills some other possibility for that nested 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 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 static member data with a particular name and of a particular type ?
- Does it have either member data or static member data with a particular name and of a particular type ?
- Does it have either a member function or static member function with a particular name and of a particular type ?

These are some of the compile-time questions which the TTI library answers. It does this by creating metafunctions, which can be used at compile-time, using C++ macros. Each of the metafunctions created returns a compile time constant bool value which answers one of the above questions at compile time. When the particular element above exists the value is 'true', or more precisely boost::mpl::true_, while if the element does not exist the value is 'false', or more precisely boost::mpl::false_. In either case the type of this value is boost::mpl::bool_.

This constant bool value, in the terminology of the Boost MPL library, is called an 'integral constant wrapper' and the metafunction generated is called a 'numerical metafunction'. The results from calling the metafunction can be passed to other metafunctions for type selection, the most popular of these being the boolean-valued operators in the Boost MPL library.

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 macro 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 a more complicated macro form, which I call the complex macro 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 macro form and a corresponding complex macro 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 always be used instead. The complex form is useful whenever using the simple form could create a duplicate name in the same name space, thereby violating the C++ one definition rule.

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 rule 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' name. 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 rule 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 be 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:

- 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 metafunction reuse 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 signature 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 signature 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 element 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 with the metafunction name in the current scope should be considered out of bounds for the programmer using TTI.

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 create metafunctions which have the exact same functionality.

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. All of the metafunctions also have a nested type called 'type', which for each one is the type of the boolean constant value. This is always boost::mpl::bool_.

Table 2. TTI Macro Metafunctions

Inner Element	Macro	Template	Specific Header File
Type	BOOST_TTI_HAS_TYPE(name)	has_type_'name' class TTI_T = enclosing type	has_type.hpp
Type with lambda expression	BOOST_TTI_HAS_TYPE(name)	has_type_'name' class TTI_T = enclosing type class TTI_U = lambda expression invoked with the inner type and returning a boolean constant	has_type.hpp
Class Template (using variadic macros)	BOOST_TTI_HAS_TEMPLATE(name)	has_template_'name' class TTI_T = enclosing type All of the template parameters must be template type parameters ('class' or 'typename' parameters)	has_template.hpp
Class Template (not using variadic macros)	BOOST_TTI_HAS_TEMPLATE(name,BOOST_PP_NIL)	has_template_'name' class TTI_T = enclosing type All of the template parameters must be template type parameters ('class' or 'typename' parameters)	has_template.hpp
Class Template with params (using variadic macros)	BOOST_TTI_HAS_TEMPLATE(name,...[a])	has_template_'name' class TTI_T = enclosing type	has_template.hpp
Class Template with params	BOOST_TTI_HAS_TEMPLATE(name,ppArray[b])	has_template_'name' class TTI_T = enclosing type	has_template.hpp
Member data	BOOST_TTI_HAS_MEMBER_DATA(name)	has_member_data_'name' class TTI_T = enclosing type OR pointer to member data ('MemberData_Type Enclosing_Type::*') class TTI_R = (optional) data type If the first parameter is the pointer to member data this must not be specified.	has_member_data.hpp

Inner Element	Macro	Template	Specific Header File
Member function	BOOST_TTI_HAS_MEMBER_FUNCTION(name)	has_member_function_'name' class TTI_T = enclosing type OR pointer to member function ('Return_Type Enclosing_Type::* (Zero or more comma-separated parameter types)') class TTI_R = (optional) return type if the first parameter is the enclosing type. If the first parameter is the pointer to member function this must not be specified. class TTI_FS = (optional) function parameter types as a Boost MPL forward sequence. If the first parameter is the pointer to member function this must not be specified. If there are no function parameters this does not have to be specified. Defaults to boost::mpl::vector<>. class TTI_TAG = (optional) Boost function_types tag type. If the first parameter is the pointer to member function this must not be specified. Defaults to boost::function_types::null_tag.	has_member_function.hpp
Static member data	BOOST_TTI_HAS_STATIC_MEMBER_DATA(name)	has_static_member_data_'name' class TTI_T = enclosing type class TTI_Type = data type	has_static_member_data.hpp

Inner Element	Macro	Template	Specific Header File
Static member function	BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION(name)	has_static_member_function_'name' class TTI_T = enclosing type class TTI_R = return type OR function type ('Return_Type (Zero or more comma-separated parameter types)') class TTI_FS = (optional) function parameter types as a Boost MPL forward sequence. If the second parameter is the function type this must not be specified. If there are no function parameters, this does not have to be specified. Defaults to boost::mpl::vector<>. class TTI_TAG = (optional) Boost function_types tag type. If the second parameter is the function type this must not be specified. Defaults to boost::function_types::null_tag.	has_static_member_function.hpp
Data, either member data or static member data	BOOST_TTI_HAS_DATA(name)	has_data_'name' class TTI_T = enclosing type class TTI_Type = data type	has_data.hpp
Function, either member function or static member function	BOOST_TTI_HAS_FUNCTION(name)	has_function_'name' class TTI_T = enclosing type class TTI_R = return type class TTI_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 TTI_TAG = (optional) Boost function_types tag type. Defaults to boost::function_types::null_tag.	has_function.hpp

[a] The template parameters as variadic data.

[b] The template parameters as the tuple part of the PP array.

Introspecting an inner type

The TTI macro `BOOST_TTI_HAS_TYPE` introspects a nested type of a class.

The BOOST_TTI_HAS_TYPE macro takes a single parameter which is the name of an inner type whose existence the programmer wants to check. The macro generates a metafunction called 'has_type_'name_of_inner_type'.

The main purpose of the generated metafunction is to check for the existence by name of the inner type. The metafunction can also be used to invoke an MPL lambda expression which is passed the inner type. One of the most common usages of the added functionality is to check whether or not the inner type is a typedef for another type.

The metafunction is invoked by passing it the enclosing type to introspect. A second type may be passed to the metafunction, an MPL lambda expression taking the inner type and returning a boolean constant.

The metafunction returns a single type called 'type', which is a boost::mpl::bool_. As a convenience the metafunction returns the value of this type directly as a compile time bool constant called 'value'. This value is true or false depending on whether the inner type exists or not.

If a second optional type is passed, this type must be an MPL lambda expression and the expression will be invoked only if the inner type exists. In that case the metafunction returns true or false depending on whether the lambda expression returns true or false. If the inner type does not exist, the lambda expression, even if specified, is never invoked and the metafunction returns false.

Generating the metafunction

You generate the metafunction by invoking the macro with the name of an inner type:

```
BOOST_TTI_HAS_TYPE(AType)
```

generates a metafunction called 'has_type_AType' in the current scope.

Invoking the metafunction

You invoke the metafunction by instantiating the template with an enclosing type to introspect and, optionally, an MPL lambda expression. A return value called 'value' is a compile time bool constant.

```
has_type_AType<Enclosing_Type>::value
has_type_AType<Enclosing_Type,ALambdaExpression>::value
```

Examples

First we generate metafunctions for various inner type names:

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

BOOST_TTI_HAS_TYPE(MyTypeDef)
BOOST_TTI_HAS_TYPE(AType)
BOOST_TTI_HAS_TYPE(ATypeDef)
BOOST_TTI_HAS_TYPE(MyType)
```

Next let us create some user-defined types we want to introspect.

```
struct Top
  {
  typedef int MyTypeDef;
  struct AType { };
  };
struct Top2
  {
  typedef long ATypeDef;
  struct MyType { };
  };
```

Finally we invoke our metafunction and return our value.

```
has_type_MyTypeDef<Top>::value;  // true
has_type_MyTypeDef<Top2>::value; // false

has_type_AType<Top>::value;  // true
has_type_AType<Top2>::value; // false

has_type_ATypeDef<Top>::value;  // false
has_type_ATypeDef<Top2>::value; // true

has_type_MyType<Top>::value;  // false
has_type_MyType<Top2>::value; // true
```

Examples - using lambda expressions

We can further invoke our metafunction with a second type, which is an MPL lambda expression.

An MPL lambda expression, an extremely useful technique in template metaprogramming, allows us to pass a metafunction to other metafunctions. The metafunction we pass can be in the form of a placeholder expression or a metafunction class. In our case the metafunction passed to our has_type_'name_of_inner_type' metafunction as a lambda expression must return a boolean constant expression.

Example - using a lambda expression with a placeholder expression

We will first illustrate the use of a lambda expression in the form of a placeholder expression being passed as the second template parameter to our has_type_'name_of_inner_type' metafunction. A popular and simple placeholder expression we can use is 'boost::is_same<_1,SomeType>' to check if the inner type found is a particular type. This is particularly useful when the inner type is a typedef for some other type.

First we include some more header files and a using declaration for convenience.

```
#include <boost/mpl/placeholders.hpp
#include <boost/type_traits/is_same.hpp
using namespace boost::mpl::placeholders;
```

Next we invoke our metafunction:

```
has_type_MyTypeDef<Top,boost::is_same<_1,int> >::value; // true
has_type_MyTypeDef<Top,boost::is_same<_1,long> >::value; // false

has_type_ATypeDef<Top2,boost::is_same<_1,int> >::value; // false
has_type_ATypeDef<Top2,boost::is_same<_1,long> >::value; // true
```

Example - using a lambda expression with a metafunction class

We will next illustrate the use of a lambda expression in the form of a metafunction class being passed as the second template parameter to our has_type_'name_of_inner_type' metafunction.

A metafunction class is a type which has a nested class template called 'apply'. For our metafunction class example we will check if the inner type is a built-in integer type. First let us write out metafunction class:

```
#include <boost/type_traits/is_integral.hpp>

class OurMetafunctionClass
  {
  template<class T> struct apply :
      boost::is_integral<T>
      {
      };
  };
```

Now we can invoke our metafunction:

```
has_type_MyTypeDef<Top,OurMetafunctionClass>::value; // true
has_type_AType<Top,OurMetafunctionClass>::value; // false

has_type_ATypeDef<Top2,OurMetafunctionClass>::value; // true
has_type_MyType<Top2,OurMetafunctionClass>::value; // true
```

Metafunction re-use

The macro encodes only the name of the inner type for which we are searching and the fact that we are introspecting for an inner type within an enclosing type.

Because of this, once we create our metafunction for introspecting an inner type by name, we can reuse the metafunction for introspecting any enclosing type, having any inner type, for that name.

Furthermore since we have only encoded the name of the inner type for which we are introspecting, we can not only introspect for that inner type by name but add different lambda expressions to inspect that inner type for whatever we want to find out about it using the same metafunction.

Introspecting an inner class template

Using the BOOST_TTI_HAS_TEMPLATE macro

The TTI macro `BOOST_TTI_HAS_TEMPLATE` introspects an inner class template of a class. The macro must specify, at the least, the name of the class template to introspect.

Two forms of introspection

There are two general forms of template introspection which can be used. The first is to find a class template with any number of only template type parameters (template parameters starting with `class` or `typename`). In this form only the name of the class template needs to be specified when invoking the macro. We will call this form of the macro the `template type parameters` form. An example of a class template of this form which could be successfully introspected would be:

```
template<class X,typename Y,class Z,typename T> class AClassTemplate { /* etc. */ };
```

The second is to find a class template with specific template parameters. In this form both the name of the class template and the template parameters are passed to the macro.

We will call this form of the macro the `specific parameters` form. An example of a class template of this form which could be successfully introspected would be:

```
template<class X, template<class> class Y, int Z> BClassTemplate { /* etc. */ };
```

When using the specific form of the macro, there are two things which need to be understood when passing the template parameters to the macro. First, the actual names of the template parameters passed are irrelevant. They can be left out completely or be different from the names in the nested class template itself. Second, the use of 'typename' or 'class', when referring to a template type parameter, is completely interchangeable, as it is in the actual class template itself.

Variadic and non-variadic macro usage

When using the BOOST_TTI_HAS_TEMPLATE macro we distinguish between compilers supporting variadic macros or not supporting variadic macros.

The programmer can always tell whether or not the compiler supports variadic macros by checking the value of the macro BOOST_PP_VARIADIC after including the necessary header file `boost/tti/has_template.hpp` in order to use the BOOST_TTI_TEMPLATE macro. A value of 1 indicates the compiler supports variadic macros while a value of 0 indicates the compiler does not support variadic macros.

Modern C++ compilers, in supporting the latest C++11 standard, normally support variadic macros. Even before the latest C++11 standard a number of C++ compilers already supported variadic macros. If you feel your compiler supports variadic macros and BOOST_PP_VARIADIC is 0 even after including `boost/tti/has_template.hpp`, you can predefine BOOST_PP_VARIADIC to 1 before including `boost/tti/has_template.hpp`.

Non-variadic macro usage

We start with syntax for compilers not supporting variadic macros since this syntax can also be used by compilers which do support variadic macros. The form for non-variadic macros always takes two macro parameters. The first macro parameter is always the name of the class template you are trying to introspect.

The second macro parameter, when using the `specific parameters` form of the macro, is the template parameters in the form of a Boost preprocessor library array data type. When using the `template type parameters` form of the macro the second macro parameter is BOOST_PP_NIL. If the second parameter is neither a Boost preprocessor library array data type or BOOS_PP_NIL you will get a compiler error if your compiler only supports non-variadic macros.

The non-variadic macro form for introspecting the class templates above using the `template type parameters` form would be:

```
BOOST_TTI_TEMPLATE(AClassTemplate,BOOST_PP_NIL)
BOOST_TTI_TEMPLATE(BClassTemplate,BOOST_PP_NIL)
```

Invoking the metafunction in the second case would always fail since the BClassTemplate does not have all template type parameters.

The non-variadic macro form for introspecting the class templates above using the `specific parameters` form would be:

```
BOOST_TTI_TEMPLATE(AClassTemplate,(4,(class,typename,class,typename)))
BOOST_TTI_TEMPLATE(BClassTemplate,(3,(class, template<class> class, int)))
```

You need to be careful using the non-variadic `specific parameters` form to specify the correct number of array parameters. This can sometimes be tricky if you have a template template parameter, or a non-type template parameter which has parentheses surrounding part of the type specification. In the latter case, when parentheses surround a comma (','), do not count that as creating another Boost PP array token. Two examples:

```
template<void (*FunctionPointer)(int,long)> class CClassTemplate { /* etc. */ };
template<template<class,class> class T> class DClassTemplate { /* etc. */ };

BOOST_TTI_TEMPLATE(CClassTemplate,(1,(void (*)(int,long))))
BOOST_TTI_TEMPLATE(DClassTemplate,(2,(template<class,class> class)))
```

In the case of using the macro to introspect CClassTemplate the number of Boost PP array parameters is 1, even though there is a comma separating the tokens in `void (*FunctionPointer)(int,long)`. This is because the comma is within parentheses.

In the case of using the macro to introspect DClassTemplate the number of Boost PP array parameters is 2, because there is a comma separating the tokens in `template<class,class> class T`.

Variadic macro usage

Having the ability to use variadic macros makes the syntax for using BOOST_TTI_TEMPLATE easier to specify in both the `template type parameters` form and the `specific parameters` form of using the macro. This is because variadic macros can take a variable number of parameters. When using the variadic macro form the first macro parameter is always the name of the class template you are trying to introspect. You only specify further parameters when using the `specific parameters` form of the macro, in which case the further parameters to the macro are the specific template parameters.

Introspecting the first class template above using the `template type parameters` form the variadic macro would be:

```
BOOST_TTI_TEMPLATE(AClassTemplate)
```

Introspecting the other class templates above using the `specific parameters` form the variadic macros would be:

```
BOOST_TTI_TEMPLATE(BClassTemplate,class,template<class> class, int)
BOOST_TTI_TEMPLATE(CClassTemplate,void (*)(int,long))
BOOST_TTI_TEMPLATE(DClassTemplate,template<class,class> class)
```

Here we have no problem with counting the number of tuple tokens for the Boost PP array, nor do we have to specify BOOST_PP_NIL if we are using the `template type parameters` form. Also for the specific parameters form we simply use the template parameters as the remaining tokens of the variadic macro.

The resulting metafunction

Using either form of the macro, whether using variadic or non-variadic syntax, the macro generates a metafunction called 'has_template_'name_of_inner_class_template'.

The metafunction can be invoked by passing it the enclosing type to introspect.

The metafunction returns a single type called 'type', which is a boost::mpl::bool_. As a convenience the metafunction returns the value of this type directly as a compile time bool constant called 'value'. This is true or false depending on whether the inner class template exists or not.

Using the has_template_(xxx) metafunction

Generating the metafunction

You generate the metafunction by invoking the macro with the name of an inner class template:

```
// `template type parameters` form

BOOST_TTI_HAS_TEMPLATE(AClassTemplate,BOOST_PP_NIL) // non-variadic macro
BOOST_TTI_HAS_TEMPLATE(AClassTemplate)              // variadic macro

// `specific parameters` form

BOOST_TTI_HAS_TEMPLATE(AClassTemplate,(2,(class,int))) // non-variadic macro
BOOST_TTI_HAS_TEMPLATE(AClassTemplate,class,int)       // variadic macro
```

generates a metafunction called 'has_template_AClassTemplate' in the current scope.

If you want to introspect the same class template name using both the `template type parameters` form and the `specific parameters` form you will have the problem that you will be generating a metafunction of the same name and violating the C++ ODR rule. In this particular case you can use the alternate BOOST_TTI_TRAIT_HAS_TEMPLATE macro to name the particular metafunction which will be generated.

Invoking the metafunction

You invoke the metafunction by instantiating the template with an enclosing type to introspect. A return value called 'value' is a compile time bool constant.

```
has_template_AType<Enclosing_Type>::value
```

Examples

First we generate metafunctions for various inner class template names:

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

// Using variadic macro, `template type parameters`

BOOST_TTI_HAS_TEMPLATE(Template1)
BOOST_TTI_HAS_TEMPLATE(Template2)
BOOST_TTI_HAS_TEMPLATE(Template3)
BOOST_TTI_HAS_TEMPLATE(Template4)
BOOST_TTI_HAS_TEMPLATE(Template5)

// or using non-variadic macro, `template type parameters`

BOOST_TTI_HAS_TEMPLATE(Template1,BOOST_PP_NIL)
BOOST_TTI_HAS_TEMPLATE(Template2,BOOST_PP_NIL)
BOOST_TTI_HAS_TEMPLATE(Template3,BOOST_PP_NIL)
BOOST_TTI_HAS_TEMPLATE(Template4,BOOST_PP_NIL)
BOOST_TTI_HAS_TEMPLATE(Template5,BOOST_PP_NIL)

// Using variadic macro, `specific parameters`

BOOST_TTI_HAS_TEMPLATE(Template6,class,int)
BOOST_TTI_HAS_TEMPLATE(Template7,typename,template<class,class> struct,long)
BOOST_TTI_HAS_TEMPLATE(Template8,double,typename)
BOOST_TTI_HAS_TEMPLATE(Template9,typename,class,typename,class,typename,short)

// or using non-variadic macro, `specific parameters`

BOOST_TTI_HAS_TEMPLATE(Template6,(2,(class,int)))
BOOST_TTI_HAS_TEMPLATE(Template7,(4,(typename,template<class,class> struct,long)))
BOOST_TTI_HAS_TEMPLATE(Template8,(2,(double,typename)))
BOOST_TTI_HAS_TEMPLATE(Template9,(6,(typename,class,typename,class,typename,short)))
```

Next let us create some user-defined types we want to introspect.

```
struct Top
  {
  template <class X> struct Template1 { };
  template <typename A,typename B,typename C> class Template2 { };
  template <typename A,typename B,typename C,int D> class Template3 { };
  };
struct Top2
  {
  template <typename A,typename B,typename C,class D> class Template3 { };
  template <class X,typename Y> struct Template4 { };
  template <typename A,class B,typename C,class D,typename E> class Template5 { };
  };
struct Top3
  {
  template <class X,int Y> struct Template6 { };
  template <typename A,template<class,class> struct B,long C> class Template7 { };
  };
struct Top4
  {
  template <double X,typename Y> struct Template8 { };
  template <typename A,class B,typename C,class D,typename E,short F> class Template9 { };
  };
```

Finally we invoke our metafunction and return our value. This all happens at compile time, and can be used by programmers doing compile time template metaprogramming.

```
has_template_Template1<Top>::value; // true
has_template_Template1<Top2>::value; // false

has_template_Template2<Top>::value; // true
has_template_Template2<Top2>::value; // false

has_template_Template3<Top>::value; // false, not all typename/class template parameters
has_template_Template3<Top2>::value; // true

has_template_Template4<Top>::value; // false
has_template_Template4<Top2>::value; // true

has_template_Template5<Top>::value; // false
has_template_Template5<Top2>::value; // true

has_template_Template6<Top3>::value; // true
has_template_Template6<Top4>::value; // false

has_template_Template7<Top3>::value; // true
has_template_Template7<Top4>::value; // false

has_template_Template8<Top3>::value; // false
has_template_Template8<Top4>::value; // true

has_template_Template9<Top3>::value; // false
has_template_Template9<Top4>::value; // true
```

Metafunction re-use

The macro encodes the name of the inner class template for which we are searching, the fact that we are introspecting for a class template within an enclosing type, and optionally the template parameters for that class template.

Once we create our metafunction for introspecting an inner class template by name, we can reuse the metafunction for introspecting any enclosing type, having any inner class template, for that name.

However we need to understand that we are restricted in our reuse of the metafunction by whether we originally use the template type parameters form or the specific form. In either case we are always introspecting an inner class template which matches that form. In the case of the template type parameters form, any inner class template for which we are introspecting must have all template type parameters, as well as the correct name. In the case of the specific parameters form, any inner class template for which we are introspecting must have template parameters which match the specific template parameters passed to the macro, as well as the correct name.

Introspecting member data

The TTI macro `BOOST_TTI_HAS_MEMBER_DATA` introspects member data of a class.

BOOST_TTI_HAS_MEMBER_DATA macro takes a single parameter which is the name of an inner member data whose existence the programmer wants to check. The macro generates a metafunction called 'has_member_data_'name_of_inner_member_data'.

The metafunction can be invoked in two different ways.

The first way is by passing it two parameters. The first parameter is the enclosing type to introspect and the second parameter is the type of the member data.

The second way is by passing it a single parameter, which is a pointer to member type. This type has the form of:

```
MemberData_Type Enclosing_Type::*
```

The metafunction returns a single type called 'type', which is a boost::mpl::bool_. As a convenience the metafunction returns the value of this type directly as a compile time bool constant called 'value'. This value is true or false depending on whether the inner member data, of the specified type, exists or not.

Generating the metafunction

You generate the metafunction by invoking the macro with the name of an inner member data:

```
BOOST_TTI_HAS_MEMBER_DATA(AMemberData)
```

generates a metafunction called 'has_member_data_AMemberData' in the current scope.

Invoking the metafunction

You invoke the metafunction by instantiating the template with an enclosing type to introspect and the type of the member data, or by instantiating the template with a pointer to member data type. The return value called 'value' is a compile time bool constant telling you whether or not the member data . exists.

```
has_member_data_AMemberData<Enclosing_Type,MemberData_Type>::value

OR

has_member_data_AMemberData<MemberData_Type Enclosing_Type::*>::value
```

Examples

First we generate metafunctions for various inner member data names:

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

BOOST_TTI_HAS_MEMBER_DATA(data1)
BOOST_TTI_HAS_MEMBER_DATA(data2)
BOOST_TTI_HAS_MEMBER_DATA(data3)
```

Next let us create some user-defined types we want to introspect.

```
struct AClass
  {
  };
struct Top
  {
  int data1;
  AClass * data2;
  };
struct Top2
  {
  long data1;
  Top data3;
  };
```

Finally we invoke our metafunction and return our value. This all happens at compile time, and can be used by programmers doing compile time template metaprogramming.

We will show both forms in the following examples. Both forms are completely interchangeable as to the result desired.

```
has_member_data_data1<Top,int>::value; // true
has_member_data_data1<Top,long>::value; // false

has_member_data_data1<Top2,int>::value; // false
has_member_data_data1<long Top2::*>::value; // true

has_member_data_data2<AClass * Top::*>::value; // true
has_member_data_data2<Top,int *>::value; // false

has_member_data_data3<int Top2::*>::value; // false
has_member_data_data3<Top Top2::*>::value; // true;
```

Metafunction re-use

The macro encodes only the name of the member data for which we are searching and the fact that we are introspecting for member data within an enclosing type.

Because of this, once we create our metafunction for introspecting an inner member data by name, we can reuse the metafunction for introspecting any enclosing type, having any inner member data type, for that name.

Introspecting member function

The TTI macro `BOOST_TTI_HAS_MEMBER_FUNCTION` introspects a member function of a class.

BOOST_TTI_HAS_MEMBER_FUNCTION takes a single parameter which is the name of an inner member function whose existence the programmer wants to check. The macro generates a metafunction called 'has_member_function_'name_of_inner_member_function'.

The metafunction can be invoked in two different ways.

The first way of invoking the metafunction is by passing it the enclosing type to introspect and a signature for the member function as a series of separate template arguments. The signature for the member function consists of the template arguments of a return type, of optional parameter types in the form of a boost::mpl forward sequence of types, and of an optional Boost FunctionTypes tag type. A typical boost::mpl forward sequence of types is a boost::mpl::vector<>.

The optional Boost FunctionTypes tag type may be used to specify cv-qualification. This means you can add 'const', 'volatile', or both by specifying an appropriate tag type. An alternate to using the tag type is to specify the enclosing type as 'const', 'volatile', or both. As an example if you specify the tag type as 'boost::function_types::const_qualified' or if you specify the enclosing type as 'const T', the member function which you are introspecting must be a const function.

The second way of invoking the metafunction is by passing it a single parameter, which is a pointer to member function. This type has the form of:

```
Return_Type Enclosing_Type::* ( Parameter_Types ) cv_qualifier(s)
```

where the Parameter_Types may be empty, or a comma-separated list of parameter types if there are more than one parameter type. The cv-qualifier may be 'const', 'volatile', or 'const volatile'.

The metafunction returns a single type called 'type', which is a boost::mpl::bool_. As a convenience the metafunction returns the value of this type directly as a compile time bool constant called 'value'. This 'value' is true or false depending on whether the inner member function, of the specified signature, exists or not.

Generating the metafunction

You generate the metafunction by invoking the macro with the name of an inner member function:

```
BOOST_TTI_HAS_MEMBER_FUNCTION(AMemberFunction)
```

generates a metafunction called 'has_member_function_AMemberFunction' in the current scope.

Invoking the metafunction

You invoke the metafunction by instantiating the template with an enclosing type to introspect and the signature of the member function as a series of template parameters. Alternatively you can invoke the metafunction by passing it a single type which is a pointer to member function.

A return value called 'value' is a compile time bool constant.

```
has_member_function_AMemberFunction
  <
  Enclosing_Type,
  MemberFunction_ReturnType,
  boost::mpl::vector<MemberFunction_ParameterTypes>, // optional, can be any mpl forward sequence
  boost::function_types::SomeTagType                 // optional, can be any FunctionTypes tag type
  >::value

OR

has_member_function_AMemberFunction
  <
  MemberFunction_ReturnType Enclosing_Type::* (MemberFunction_ParameterTypes) optional_cv_quali↵
fication
  >::value
```

Examples

First we generate metafunctions for various inner member function names:

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

BOOST_TTI_HAS_MEMBER_FUNCTION(function1)
BOOST_TTI_HAS_MEMBER_FUNCTION(function2)
BOOST_TTI_HAS_MEMBER_FUNCTION(function3)
```

Next let us create some user-defined types we want to introspect.

```
struct AClass
  {
  };
struct Top
  {
  int function1();
  AClass function2(double,short *);
  };
struct Top2
  {
  long function2(Top &,int,bool,short,float);
  Top * function3(long,int,AClass &);
  };
```

Finally we invoke our metafunction and return our value. This all happens at compile time, and can be used by programmers doing compile time template metaprogramming.

We will show both forms in the following examples. Both forms are completely interchangeable as to the result desired.

```
has_member_function_function1<Top,int>::value; // true
has_member_function_function1<Top,int,boost::mpl::vector<> >::value; // true
has_member_function_function1<Top2,int>::value; // false

has_member_function_function2<AClass Top::* (double,short *)>::value; // true
has_member_function_function2<AClass Top2::* (double,short *)>::value; // false
has_member_function_function2<long Top2::* (Top &,int,bool,short,float)>::value; // true

has_member_function_function3<int Top2::* ()>::value; // false
has_member_function_function3<Top2,Top *,boost::mpl::vector<long,int,AClass &> >::value; // true;
```

Metafunction re-use

The macro encodes only the name of the member function for which we are searching and the fact that we are introspecting for a member function within an enclosing type.

Because of this, once we create our metafunction for introspecting a member function by name, we can reuse the metafunction for introspecting any enclosing type, having any member function, for that name.

Introspecting static member data

The TTI macro `BOOST_TTI_HAS_STATIC_MEMBER_DATA` introspects static member data of a class.

BOOST_TTI_HAS_STATIC_MEMBER_DATA macro takes a single parameter which is the name of an inner static member data whose existence the programmer wants to check. The macro generates a metafunction called 'has_static_member_data_'name_of_inner_static_member_data'.

The metafunction can be invoked by passing it the enclosing type to introspect and the type of the static member data.

The metafunction returns a single type called 'type', which is a boost::mpl::bool_. As a convenience the metafunction returns the value of this type directly as a compile time bool constant called 'value'. This is true or false depending on whether the inner static member data, of the specified type, exists or not.

Generating the metafunction

You generate the metafunction by invoking the macro with the name of an inner static member data:

```
BOOST_TTI_HAS_STATIC_MEMBER_DATA(AStaticMemberData)
```

generates a metafunction called 'has_static_member_data_AStaticMemberData' in the current scope.

Invoking the metafunction

You invoke the metafunction by instantiating the template with an enclosing type to introspect and the type of the static member data. A return value called 'value' is a compile time bool constant.

```
has_static_member_data_AStaticMemberData<Enclosing_Type,StaticMemberData_Type>::value
```

Examples

First we generate metafunctions for various inner member data names:

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

BOOST_TTI_HAS_STATIC_MEMBER_DATA(data1)
BOOST_TTI_HAS_STATIC_MEMBER_DATA(data2)
BOOST_TTI_HAS_STATIC_MEMBER_DATA(data3)
```

Next let us create some user-defined types we want to introspect.

```
struct AClass
  {
  };
struct Top
  {
  static int data1;
  static AClass * data2;
  };
struct Top2
  {
  static long data1;
  static Top data3;
  };
```

Finally we invoke our metafunction and return our value. This all happens at compile time, and can be used by programmers doing compile time template metaprogramming.

```
has_static_member_data_data1<Top,int>::value; // true
has_static_member_data_data1<Top,long>::value; // false
has_static_member_data_data1<Top2,int>::value; // false
has_static_member_data_data1<Top2,long>::value; // true

has_static_member_data_data2<Top,AClass *>::value; // true
has_static_member_data_data2<Top,int *>::value; // false

has_static_member_data_data3<Top2,int>::value; // false
has_static_member_data_data3<Top2,Top>::value; // true;
```

Metafunction re-use

The macro encodes only the name of the static member data for which we are searching and the fact that we are introspecting for static member data within an enclosing type.

Because of this, once we create our metafunction for introspecting an inner static member data by name, we can reuse the metafunction for introspecting any enclosing type, having any inner static member data type, for that name.

Introspecting static member function

The TTI macro `BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION` introspects a static member function of a class.

BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION takes a single parameter which is the name of an inner static member function whose existence the programmer wants to check. The macro generates a metafunction called 'has_static_member_function_'name_of_inner_static_member_function'.

The metafunction can be invoked in two different ways.

The first way is by passing it the enclosing type to introspect and a signature for the static member function as separate template arguments. The signature for the static member function consists of a return type, optional parameter types in the form of a boost::mpl forward sequence of types, and an optional Boost FunctionTypes tag type. A typical boost::mpl forward sequence of types is a boost::mpl::vector<>.

The second way is by passing it the enclosing type to introspect and a signature for the static member function as a function. The function has the form of:

```
Return_Type ( Parameter_Types )
```

where the Parameter_Types may be empty, or a comma-separated list of parameter types if there are more than one parameter type.

The metafunction returns a single type called 'type', which is a boost::mpl::bool_. As a convenience the metafunction returns the value of this type directly as a compile time bool constant called 'value'. This is true or false depending on whether the inner static member function, of the specified signature, exists or not.

Generating the metafunction

You generate the metafunction by invoking the macro with the name of an inner static member function:

```
BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION(AStaticMemberFunction)
```

generates a metafunction called 'has_static_member_function_AStaticMemberFunction' in the current scope.

Invoking the metafunction

You invoke the metafunction by instantiating the template with an enclosing type to introspect and the signature of the static member function as a series of template parameters. Alternatively you can invoke the metafunction by passing it an enclosing type and the signature of the static member function as a single function type.

A return value called 'value' is a compile time bool constant.

```
has_static_member_function_AStaticMemberFunction
  <
  Enclosing_Type,
  StaticMemberFunction_ReturnType,
  boost::mpl::vector<StaticMemberFunction_ParameterTypes>, // optional, can be any mpl forward sequence
  boost::function_types::SomeTagType                       // optional, can be any FunctionTypes tag type
  >::value

OR

has_static_member_function_AStaticMemberFunction
  <
  Enclosing_Type,
  Return_Type ( Parameter_Types )
  >::value
```

Examples

First we generate metafunctions for various inner static member function names:

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

BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION(function1)
BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION(function2)
BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION(function3)
```

Next let us create some user-defined types we want to introspect.

```
struct AClass { };
struct Top
  {
  static int function1();
  static AClass function2(double,short *);
  };
struct Top2
  {
  static long function2(Top &,int,bool,short,float);
  static Top * function3(long,int,AClass &);
  };
```

Finally we invoke our metafunction and return our value. This all happens at compile time, and can be used by programmers doing compile time template metaprogramming.

We will show both forms in the following examples. Both forms are completely interchangeable as to the result desired.

```
has_static_member_function_function1<Top,int>::value; // true
has_static_member_function_function1<Top,int ()>::value; // true
has_static_member_function_function1<Top2,int>::value; // false

has_static_member_function_function2<Top,AClass,boost::mpl::vector<double,short *> >::value; // true
has_static_member_function_function2<Top2,AClass,boost::mpl::vector<double,short *> >::value; // false
has_static_member_function_function2<Top2,long (Top &,int,bool,short,float)>::value; // true

has_static_member_function_function3<Top2,int ()>::value; // false
has_static_member_function_function3<Top2,Top * (long,int,AClass &)>::value; // true;
```

Metafunction re-use

The macro encodes only the name of the static member function for which we are searching and the fact that we are introspecting for a static member function within an enclosing type.

Because of this, once we create our metafunction for introspecting a static member function by name, we can reuse the metafunction for introspecting any enclosing type, having any static member function, for that name.

Introspecting inner data

The TTI macro BOOST_TTI_HAS_DATA introspects the data of a class. The data can be member data or static member data.

BOOST_TTI_HAS_DATA macro takes a single parameter, which is the name of an inner member data or static member data, whose existence the programmer wants to check. The macro generates a metafunction called 'has_data_'name_of_inner_data'.

The metafunction can be invoked by passing it the enclosing type to introspect and the type of the data.

The metafunction returns a single type called 'type', which is a boost::mpl::bool_. As a convenience the metafunction returns the value of this type directly as a compile time bool constant called 'value'. This is true or false depending on whether the inner data, of the specified type, exists or not.

Generating the metafunction

You generate the metafunction by invoking the macro with the name of an inner data member:

```
BOOST_TTI_HAS_DATA(AData)
```

generates a metafunction called 'has_data_AStaticMemberData' in the current scope.

Invoking the metafunction

You invoke the metafunction by instantiating the template with an enclosing type to introspect and the type of the data. A return value called 'value' is a compile time bool constant.

```
has_data_AData<Enclosing_Type,Data_Type>::value
```

Examples

First we generate metafunctions for various inner data names:

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

BOOST_TTI_HAS_DATA(data1)
BOOST_TTI_HAS_DATA(data2)
BOOST_TTI_HAS_DATA(data3)
```

Next let us create some user-defined types we want to introspect.

```
struct AClass
  {
  };
struct Top
  {
  int data1;
  static AClass * data2;
  };
struct Top2
  {
  static long data1;
  Top data3;
  };
```

Finally we invoke our metafunction and return our value. This all happens at compile time, and can be used by programmers doing compile time template metaprogramming.

```
has_data_data1<Top,int>::value; // true
has_data_data1<Top,long>::value; // false
has_data_data1<Top2,int>::value; // false
has_data_data1<Top2,long>::value; // true

has_data_data2<Top,AClass *>::value; // true
has_data_data2<Top,int *>::value; // false

has_data_data3<Top2,int>::value; // false
has_data_data3<Top2,Top>::value; // true;
```

Metafunction re-use

The macro encodes only the name of the data for which we are searching and the fact that we are introspecting for data within an enclosing type.

Because of this, once we create our metafunction for introspecting an inner data by name, we can reuse the metafunction for introspecting any enclosing type, having any inner data type, for that name.

Introspecting an inner function

The TTI macro BOOST_TTI_HAS_FUNCTION introspects an inner function of a class. The function can be either a member function or a static member function.

BOOST_TTI_HAS_FUNCTION takes a single parameter which is the name of an inner function whose existence the programmer wants to check. The macro generates a metafunction called 'has_function_'name_of_inner_function'.

The metafunction can be invoked by passing it the enclosing type to introspect and a signature for the function as separate template arguments. The signature for the function consists of a return type, optional parameter types in the form of a boost::mpl forward sequence of types, and an optional Boost FunctionTypes tag type. A typical boost::mpl forward sequence of types is a boost::mpl::vector<>.

The metafunction returns a single type called 'type', which is a boost::mpl::bool_. As a convenience the metafunction returns the value of this type directly as a compile time bool constant called 'value'. This is true or false depending on whether the inner function, of the specified signature, exists or not.

Generating the metafunction

You generate the metafunction by invoking the macro with the name of an inner function:

```
BOOST_TTI_HAS_FUNCTION(AnInnerFunction)
```

generates a metafunction called 'has_function_AnInnerFunction' in the current scope.

Invoking the metafunction

You invoke the metafunction by instantiating the template with an enclosing type to introspect and the signature of the function as a series of template parameters.

A return value called 'value' is a compile time bool constant.

```
has_function_AnInnerFunction
  <
  Enclosing_Type,
  Function_ReturnType,
  boost::mpl::vector<Function_ParameterTypes>, // optional, can be any mpl forward sequence
  boost::function_types::SomeTagType           // optional, can be any FunctionTypes tag type
  >::value
```

Examples

First we generate metafunctions for various inner function names:

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

BOOST_TTI_HAS_FUNCTION(function1)
BOOST_TTI_HAS_FUNCTION(function2)
BOOST_TTI_HAS_FUNCTION(function3)
```

Next let us create some user-defined types we want to introspect.

```
struct AClass { };
struct Top
  {
  static int function1();
  AClass function2(double,short *);
  };
struct Top2
  {
  long function2(Top &,int,bool,short,float);
  static Top * function3(long,int,AClass &);
  };
```

Finally we invoke our metafunction and return our value. This all happens at compile time, and can be used by programmers doing compile time template metaprogramming.

```
has_function_function1<Top,int>::value; // true
has_function_function1<Top2,int>::value; // false

has_function_function2<Top,AClass,boost::mpl::vector<double,short *> >::value; // true
has_function_function2<Top2,AClass,boost::mpl::vector<double,short *> >::value; // false

has_function_function3<Top2,int>::value; // false
has_function_function3<Top2,Top *,boost::mpl::vector<long,int,AClass &> >::value; // true;
```

Metafunction re-use

The macro encodes only the name of the function for which we are searching and the fact that we are introspecting for a function within an enclosing type.

Because of this, once we create our metafunction for introspecting a function by name, we can reuse the metafunction for introspecting any enclosing type, having any function, for that name.

Nested Types

Besides the functionality of the TTI library which queries whether some inner element of a given name within a type exists, the library also includes functionality for generating a nested type if it exists, else a marker type if it does not exist. By marker type is meant a type either internally created by the library, with no functionality, or designated by the end-user to represent the same idea.

First I will explain the syntax and use of this functionality and then the reason it exists in the library.

The functionality is a metafunction created by the macro `BOOST_TTI_MEMBER_TYPE`. The macro takes a single parameter, which is the name of a nested type. We will call this our 'named type'. The macro generates a metafunction called `member_type_'named_type'` which, passed an enclosing type, returns the named type if it exists, else a marker type if it does not.

As with our other macros we can use the alternative form of the macro `BOOST_TTI_TRAIT_MEMBER_TYPE` to pass first the name of the metafunction to be generated and then the name of the 'named type'. After that the functionality of our resulting metafunction is exactly the same.

Its general explanation is given as:

Table 3. 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 class U = (optional) marker type returns = the type of 'name' if it exists, else a marker type, as the typedef 'type'. The invoked metafunction also holds the marker type as the typedef 'boost_tti_marker_type'. This is done for convenience so that the marker type does not have to be remembered.	`member_type.hpp`

The marker type is purely optional. If not specified a type internal to the TTI library, which has no functionality, is used. Unless there is a specific reason for the end-user to provide his own marker type, he should let the TTI library use its own internal marker type.

A simple example of this functionality would be:

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

BOOST_TTI_MEMBER_TYPE(ANamedType)

typedef typename member_type_ANamedType<EnclosingType>::type AType;
```

If type 'ANamedType' is a nested type of 'EnclosingType' then AType is the same type as 'ANamedType', otherwise AType is a marker type internal to the TTI library.

Now that we have explained the syntax of BOOST_TTI_MEMBER_TYPE we can now answer the question of why this functionality to create a 'type' exists when looking for a nested type of an enclosing type.

The problem

The metafunctions generated by the TTI macros all work with various 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.

This is a problem if 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 compiler 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 in conjunction with Boost MPL's selection metafunctions, we can do compile-time selection to generate the correct template code.

However this does not 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.

The solution

The solution given by BOOST_TTI_MEMBER_TYPE is that we can create a type as the return from our metafunction, which is the same type as a nested type if it exists or some other marker type if it does not, and then work with that returned type without producing a compiler error. If we had to use the 'T::InnerType' syntax to specify our type, where 'T' represents out enclosing type and 'InnerType' our nested type, and there was no nested type 'InnerType' within the enclosing type 'T, the compiler would give us an error immediately.

By using BOOST_TTI_MEMBER_TYPE we have a type to work with even when such a type really does not exist. Naturally if the type does not exist, the type which we have to work with, being a marker type, will generally not fulfill any other further functionality we want from it. This is good and will normally produce the correct results in further uses of the type when doing metafunction programming. Occasionally the TTI produced marker type, when our nested type does not exist, is not sufficient for further metafunction programming. In that rare case the end-user can produce his own marker type to be used if the nested type does not exist. In any case, whether the nested type exists, whether the TTI default supplied marker type is used, or whether an end-user marker type is used, template metaprogramming can continue without a compilation problem. Furthermore this scales better than having to constant check for nested type existence via BOOST_TTI_HAS_TYPE in complicated template metaprogramming code.

Checking if the member type exists

Once we use BOOST_TTI_MEMBER_TYPE to generate a nested type if it exists we will normally use that type in further metafunction programming. Occasionally, given the type we generate, we will want to ask if the type is really our nested type or the marker type instead. Essentially we are asking if the type generated is the marker type or not. If it is the marker type, then the type generated is not the nested type we had hoped for. If it is not the marker type, then the type generated is the nested type we had hoped for. This is easy enough to do for the template metaprogrammer but TTI makes it easier by providing either of two metafunctions to do this calculation. These two metafunctions are 'boost::tti::valid_member_type' and 'boost::tti::valid_member_metafunction':

Table 4. TTI Nested Type Macro Metafunction Existence

Inner Element	Macro	Template	Specific Header File
Type	None	`boost::tti::valid_member_type` class T = a type class U = (optional) marker type returns = true if the type exists, false if it does not. 'Existence' is determined by whether the type does not equal the marker type of BOOST_TTI_MEMBER_TYPE.	`member_type.hpp`
Type	None	`boost::tti::valid_member_metafunction` class T = a metafunction type returns = true if the return 'type' of the metafunction exists, false if it does not.'Existence' is determined by whether the return 'type' does not equal the marker type of BOOST_TTI_MEMBER_TYPE.	`member_type.hpp`

In our first metafunction, 'boost::tti::valid_member_type', the first parameter is the return 'type' from invoking the metafunction generated by BOOST_TTI_MEMBER_TYPE. If when the metafunction was invoked a user-defined marker type had been specified, then the second optional parameter is that marker type, else it is not necessary to specify the optional second template parameter. Since the marker type is saved as the nested type boost::tti::marker_type once we invoke the metafunction generated by BOOST_TTI_MEMBER_TYPE we can always use that as our second template parameter to 'boost::tti::valid_member_type' if we like.

The second metafunction, boost::tti::valid_member_metafunction, makes the process of passing our nested 'type' and our marker type a bit easier. Here the single template parameter is the invoked metafunction generated by BOOST_TTI_MEMBER_TYPE itself. It then picks out from the invoked metafunction both the return 'type' and the nested boost::tti::marker_type to do the correct calculation.

A simple example of this functionality would be:

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

struct UDMarkerType { };

BOOST_TTI_MEMBER_TYPE(ANamedType)

typedef member_type_ANamedType<EnclosingType> IMType;
typedef member_type_ANamedType<EnclosingType,UDMarkerType> IMTypeWithMarkerType;
```

then

```
boost::tti::valid_member_type<IMType::type>::value
boost::tti::valid_member_type<IMTypeWithMarkerType::type,IMTypeWithMarkerType::boost_tti_mark↵
er_type>::value
```

or

```
boost::tti::valid_member_metafunction<IMType>::value
boost::tti::valid_member_metafunction<IMTypeWithMarkerType>::value
```

gives us our compile-time result.

An extended nested type example

As an extended 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. We have to do this in the reverse order of our hypothetical 'struct T' above since the metafunction `BOOST_TTI_MEMBER_TYPE` takes its enclosing type as its template parameter.

```
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. It will only generate, if it does not exist, an eventual failure by having whatever metafunction uses such a type 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:

```
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.

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, could 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. Instead we use the functionality already provided by 'boost::tti::valid_member_type'. Using this functionality with our 'MyFindType' type above we create the nullary metafunction:

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

directly instead of replicating the same functionality with our '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 ways of using our generated metafunction when introspecting for member data, a member function, or a static member function of an enclosing type.

In the cases where we specify a composite syntax when using BOOST_TTI_HAS_MEMBER_DATA, BOOST_TTI_HAS_MEMBER_FUNCTION, or BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION, the signature for the member data, member function, or static member function is a single type. For BOOST_TTI_HAS_MEMBER_DATA the signature is a pointer to member data, for BOOST_TTI_HAS_MEMBER_FUNCTION the signature is a pointer to a member function, and for BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION the signature is divided between an enclosing type and a function in composite format. This makes for a syntactical notation which is natural to specify, but because of the notation we can not use the nested type functionality in BOOST_TTI_MEMBER_TYPE for potential parts of these composite types. If any part of this signature, which specifies a composite of various types, is invalid, a compiler time error will occur.

But in the more specific cases, when we use BOOST_TTI_HAS_MEMBER_DATA, BOOST_TTI_HAS_MEMBER_FUNCTION, and BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION, our composite type in our signatures is broken down into their individual types so that using BOOST_TTI_MEMBER_TYPE for any one of the individual types will not lead to a compile time error if the type specified does not actually exist.

A few examples will suffice.

Given known types T and U, and the supposed type Ntype as a nested type of U, we want to find out if type T has a member function whose signature is `void aMemberFunction(U::Ntype)`.

First using BOOST_TTI_HAS_MEMBER_FUNCTION using our composite form we would code:

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

BOOST_TTI_HAS_MEMBER_FUNCTION(aMemberFunction)

has_member_function_aMemberFunction<void (T::*)(U::Ntype)>::value;
```

If the nested type U::Ntype does not exist, this leads to a compiler error. We really want to avoid this situation, so let's try our alternative.

Second using BOOST_TTI_HAS_MEMBER_FUNCTION using our specific form we would code:

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

BOOST_TTI_HAS_MEMBER_TYPE(Ntype)
BOOST_TTI_HAS_MEMBER_FUNCTION(aMemberFunction)

typedef typename has_member_type_Ntype<U>::type OurType;
has_member_function_aMemberFunction<T,void,boost::mpl::vector<OurType> >::value;
```

If the nested type U::Ntype does exist and T does have a member function whose signature is `void aMemberFunction(U::Ntype)` our 'value' is true, otherwise it is false. We will never get a compiler error in this case.

As a second example we will once again use the suppositions of our first example; given known types T and U, and the supposed type Ntype as a nested type of U. But this time let us look for a static member function whose signature is `void aStaticMemberFunction(U::Ntype)`.

First using BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION using our composite form we would code:

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

BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION(aStaticMemberFunction)

has_static_member_function_aStaticMemberFunction<T,void (U::Ntype)>::value;
```

Once again if the nested type U::Ntype does not exist, this leads to a compiler error, so let's try our alternative.

Second using `BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION` using our specific form we would code:

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

BOOST_TTI_HAS_MEMBER_TYPE(Ntype)
BOOST_TTI_HAS_STATIC_MEMBER_FUNCTION(aStaticMemberFunction)

typedef typename has_member_type_Ntype<U>::type OurType;
has_static_member_function_aStaticMemberFunction<T,void,boost::mpl::vector<OurType> >::value;
```

If the nested type U::Ntype does exist and T does have a member function whose signature is `void aMemberFunction(U::Ntype)` our 'value' is true, otherwise it is false. We will never get a compiler error in this case.

An example 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 where 'T' is a type passed to the 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,class> struct AnotherMemberTemplate { };
  template <class,class,int,class,template <class> class,class,long> struct ManyParameters { };
  template <class,class,int,short,class,template <class,int> class,class> struct MoreParamet↵
ers { };

  // 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)

has_type_AnIntType
  <
  T
  >
```

Does T have a nested type called 'BType' ?

```
BOOST_TTI_HAS_TYPE(BType)

has_type_BType
  <
  T
  >
```

Type checking the typedef using a lambda expression

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

```
#include <boost/mpl/placeholders.hpp
#include <boost/type_traits/is_same.hpp
using namespace boost::mpl::placeholders;

has_type_AnIntType
  <
  T,
  boost::is_same<_1,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_PP_NIL)

has_template_AMemberTemplate
  <
  T
  >
```

Template using variadic macros

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

```
BOOST_TTI_HAS_TEMPLATE(AnotherMemberTemplate)

has_template_AnotherMemberTemplate
  <
  T
  >
```

Template with params

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

```
BOOST_TTI_HAS_TEMPLATE(MoreParameters,(8,(class,class,int,short,class,tem↵
plate <class,int> class,class)))

has_template_MoreParameters
  <
  T
  >
```

Template with params using variadic macros

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

```
BOOST_TTI_HAS_TEMPLATE(ManyParameters,class,class,int,class,template <class> class,class,long)

has_template_ManyParameters
  <
  T
  >
```

Member data

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

```
BOOST_TTI_HAS_MEMBER_DATA(IntBT)

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

Member data with composite type

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

```
BOOST_TTI_HAS_MEMBER_DATA(IntBT)

has_member_data_IntBT
  <
  AType::BType T::*
  >
```

Member function with individual types

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

```
BOOST_TTI_HAS_MEMBER_FUNCTION(IntFunction)

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(IntFunction)
has_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)

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)

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(SIntFunction)

has_static_member_function_SIntFunction
  <
  T,
  int (long,double)
  >
```

Data

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

```
BOOST_TTI_HAS_DATA(DSMember)

has_static_member_data_DSMember
  <
  T,
  short
  >
```

Function

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

```
BOOST_TTI_HAS_FUNCTION(IntFunction)

has_function_IntFunction
  <
  T,
  int,
  boost::mpl::vector<short>
  >
```

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
member_type_CType
  <
  typename
  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
member_type_CType
  <
  typename
  member_type_BType
    <
    T
    >::type
  >::type
AType;

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

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, which serves as a type. 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. Although checking if some particular instantiation of a nested function template exists at compile-time does not prove that the nested function template itself does or does not exist, since the instantiation itself may be incorrect and fail even when the nested function template exists, it provides a partial, if 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 C' is the type of the enclosing class and the 'class T' is the type of the instantiated member function template as a member function.

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.

Reference

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

```
BOOST_TTI_HAS_DATA_GEN(name)
```

Macro BOOST_TTI_HAS_DATA_GEN

BOOST_TTI_HAS_DATA_GEN — Generates the macro metafunction name for BOOST_TTI_HAS_DATA.

Synopsis

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

BOOST_TTI_HAS_DATA_GEN(name)
```

Description

name = the name of the member data.

returns = the generated macro metafunction name.

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

```
BOOST_TTI_HAS_FUNCTION_GEN(name)
```

Macro BOOST_TTI_HAS_FUNCTION_GEN

BOOST_TTI_HAS_FUNCTION_GEN — Generates the macro metafunction name for BOOST_TTI_HAS_FUNCTION.

Synopsis

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

BOOST_TTI_HAS_FUNCTION_GEN(name)
```

Description

name = the name of the static member function.

returns = the generated macro metafunction name.

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_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_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/has_data.hpp>

```
BOOST_TTI_TRAIT_HAS_DATA(trait, name)
BOOST_TTI_HAS_DATA(name)
```

Macro BOOST_TTI_TRAIT_HAS_DATA

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

Synopsis

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

BOOST_TTI_TRAIT_HAS_DATA(trait, name)
```

Description

trait = the name of the metafunction.

name = the name of the inner member to introspect.

generates a metafunction called "trait" where 'trait' is the macro parameter. template<class BOOST_TTI_TP_T,class BOOST_TTI_TP_TYPE> struct trait { static const value = unspecified; typedef mpl::bool_<true-or-false> type; }; The metafunction types and return: BOOST_TTI_TP_T = the enclosing type in which to look for our 'name' BOOST_TTI_TP_TYPE = The type of the member data or static member. returns = 'value' is true if the 'name' exists, with the correct data type, otherwise 'value' is false.

Macro BOOST_TTI_HAS_DATA

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

Synopsis

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

BOOST_TTI_HAS_DATA(name)
```

Description

name = the name of the inner member.

generates a metafunction called "has_data_name" where 'name' is the macro parameter. template<class BOOST_TTI_TP_T,class BOOST_TTI_TP_TYPE> struct has_data_name { static const value = unspecified; typedef mpl::bool_<true-or-false> type; }; The metafunction types and return: BOOST_TTI_TP_T = the enclosing type in which to look for our 'name' BOOST_TTI_TP_TYPE = The type of the member data or static member. returns = 'value' is true if the 'name' exists, with the correct data type, otherwise 'value' is false.

Header <boost/tti/has_function.hpp>

```


BOOST_TTI_TRAIT_HAS_FUNCTION(trait, name)
BOOST_TTI_HAS_FUNCTION(name)
```

Macro BOOST_TTI_TRAIT_HAS_FUNCTION

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

Synopsis

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

BOOST_TTI_TRAIT_HAS_FUNCTION(trait, name)
```

Description

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

name = the name of the inner member.

generates a metafunction called "trait" where 'trait' is the macro parameter. template<class BOOST_TTI_TP_T,class BOOST_TTI_TP_R,class BOOST_TTI_TP_FS,class BOOST_TTI_TP_TAG> struct trait { static const value = unspecified; typedef mpl::bool_<true-or-false> type; }; The metafunction types and return: BOOST_TTI_TP_T = the enclosing type in which to look for our 'name'. BOOST_TTI_TP_R = the return type of the function BOOST_TTI_TP_FS = (optional) the parameters of the function as a boost::mpl forward sequence if function parameters are not empty. BOOST_TTI_TP_TAG = (optional) a boost::function_types tag to apply to the function if the need for a tag exists. returns = 'value' is true if the 'name' exists, with the appropriate static member function type, otherwise 'value' is false.

Macro BOOST_TTI_HAS_FUNCTION

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

Synopsis

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

BOOST_TTI_HAS_FUNCTION(name)
```

Description

name = the name of the inner member.

generates a metafunction called "has_function_name" where 'name' is the macro parameter. template<class BOOST_TTI_TP_T,class BOOST_TTI_TP_R,class BOOST_TTI_TP_FS,class BOOST_TTI_TP_TAG> struct trait { static const value = unspecified; typedef mpl::bool_<true-or-false> type; }; The metafunction types and return: BOOST_TTI_TP_T = the enclosing type in which to look for our 'name'. BOOST_TTI_TP_R = the return type of the function BOOST_TTI_TP_FS = (optional) the parameters of the function as a boost::mpl forward sequence if function parameters are not empty. BOOST_TTI_TP_TAG = (optional) a boost::function_types tag to apply to the function if the need for a tag exists. returns = 'value' is true if the 'name' exists, with the appropriate function type, otherwise 'value' is false.

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.

name = the name of the inner member to introspect.

generates a metafunction called "trait" where 'trait' is the macro parameter. template<class BOOST_TTI_TP_ET,class BOOST_TTI_TP_TYPE> struct trait { static const value = unspecified; typedef mpl::bool_<true-or-false> type; }; The metafunction types and return: BOOST_TTI_TP_ET = the enclosing type in which to look for our 'name' OR The type of the member data in the form of a pointer to member data. BOOST_TTI_TP_TYPE = (optional) The type of the member data if the first parameter is the enclosing type. returns = 'value' is true if the 'name' exists, with the correct data 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.

generates a metafunction called "has_member_data_name" where 'name' is the macro parameter. template<class BOOST_TTI_TP_ET,class BOOST_TTI_TP_TYPE> struct has_member_data_name { static const value = unspecified; typedef mpl::bool_<true-or-false> type; }; The metafunction types and return: BOOST_TTI_TP_ET = the enclosing type in which to look for our 'name' OR The type of the member data in the form of a pointer to member data. BOOST_TTI_TP_TYPE = (optional) The type of the member data if the first parameter is the enclosing type. returns = 'value' is true if the 'name' exists, with the correct data 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.

generates a metafunction called "trait" where 'trait' is the macro parameter.
template<class BOOST_TTI_TP_T,class BOOST_TTI_R,class BOOST_TTI_FS,class BOOST_TTI_TAG> struct trait { static const value = unspecified; typedef mpl::bool_<true-or-false> type; }; The metafunction types and return: BOOST_TTI_TP_T = the enclosing type in which to look for our 'name' OR a pointer to member function as a single type. BOOST_TTI_TP_R = (optional) the return type of the member function if the first parameter is the enclosing type. BOOST_TTI_TP_FS = (optional) the parameters of the member function as a boost::mpl forward sequence if the first parameter is the enclosing type and the member function parameters are not empty. BOOST_TTI_TP_TAG = (optional) a boost::function_types tag to apply to the member function if the first parameter

is the enclosing type and a tag is needed. returns = 'value' is true if the 'name' exists, with the appropriate member function 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.

generates a metafunction called "has_member_function_name" where 'name' is the macro parameter. template<class BOOST_TTI_TP_T,class BOOST_TTI_TP_R,class BOOST_TTI_TP_FS,class BOOST_TTI_TP_TAG> struct has_member_function_name { static const value = unspecified; typedef mpl::bool_<true-or-false> type; }; The metafunction types and return: BOOST_TTI_TP_T = the enclosing type in which to look for our 'name' OR a pointer to member function as a single type. BOOST_TTI_TP_R = (optional) the return type of the member function if the first parameter is the enclosing type. BOOST_TTI_TP_FS = (optional) the parameters of the member function as a boost::mpl forward sequence if the first parameter is the enclosing type and the member function parameters are not empty. BOOST_TTI_TP_TAG = (optional) a boost::function_types tag to apply to the member function if the first parameter is the enclosing type and a tag is needed. returns = 'value' is true if the 'name' exists, with the appropriate member function 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.

generates a metafunction called "trait" where 'trait' is the macro parameter. The metafunction types and return: BOOST_TTI_TP_T = the enclosing type. BOOST_TTI_TP_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, with the BOOST_TTI_TP_TYPE type, within the enclosing BOOST_TTI_TP_T 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.

generates a metafunction called "has_static_member_data_name" where 'name' is the macro parameter. The metafunction types and return: BOOST_TTI_TP_T = the enclosing type. BOOST_TTI_TP_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, with the appropriate BOOST_TTI_TP_TYPE type, within the enclosing BOOST_TTI_TP_T 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.

generates a metafunction called "trait" where 'trait' is the macro parameter. template<class BOOST_TTI_TP_T,class BOOST_TTI_TP_R,class BOOST_TTI_TP_FS,class BOOST_TTI_TP_TAG> struct trait { static const value = unspecified; typedef mpl::bool_<true-or-false> type; }; The metafunction types and return: BOOST_TTI_TP_T = the enclosing type in which to look for our 'name'. BOOST_TTI_TP_R = the return type of the static member function OR the signature of a function in the form of Return_Type (Parameter_Types) BOOST_TTI_TP_FS = (optional) the parameters of the static member function as a boost::mpl forward sequence if the second parameter is a return type and the function parameters exist. BOOST_TTI_TP_TAG = (optional) a

boost::function_types tag to apply to the static member function if the second parameter is a return type and the need for a tag exists. returns = 'value' is true if the 'name' exists, with the appropriate static member function 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.

generates a metafunction called "has_static_member_function_name" where 'name' is the macro parameter. template<class BOOST_TTI_TP_T,class BOOST_TTI_TP_R,class BOOST_TTI_TP_FS,class BOOST_TTI_TP_TAG> struct trait { static const value = unspecified; typedef mpl::bool_<true-or-false> type; }; The metafunction types and return: BOOST_TTI_TP_T = the enclosing type in which to look for our 'name'. BOOST_TTI_TP_R = the return type of the static member function OR the signature of a function in the form of Return_Type (Parameter_Types) BOOST_TTI_TP_FS = (optional) the parameters of the static member function as a boost::mpl forward sequence if the second parameter is a return type and the function parameters exist. BOOST_TTI_TP_TAG = (optional) a boost::function_types tag to apply to the static member function if the second parameter is a return type and the need for a tag exists. returns = 'value' is true if the 'name' exists, with the appropriate static member function type, otherwise 'value' is false.

Header <boost/tti/has_template.hpp>

```


BOOST_TTI_TRAIT_HAS_TEMPLATE(trait, ...)
BOOST_TTI_HAS_TEMPLATE(...)
```

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, ...)
```

Description

trait = the name of the metafunction. ... = variadic parameters. The first variadic parameter is the inner class template name. Following variadic parameters are optional. If no following variadic parameters exist, then the inner class template being introspected must be all template type parameters (template parameters starting with `class` or `typename`) and any number of template type parameters can occur. If the second variadic parameter is BOOST_PP_NIL and no other variadic parameter is given, then just as in the previous case the inner class template being introspected must be all template type parameters (template parameters starting with `class` or `typename`) and any number of template type parameters can occur. This form is allowed in order to be consistent with using the

non-variadic form of this macro. If the second variadic parameter is a Boost preprocessor library array and no other variadic parameter is given, then the inner class template must have its template parameters matching the sequence in the tuple portion of the Boost PP array. This form is allowed in order to be consistent with using the non-variadic form of this macro. Otherwise the inner class template must have its template parameters matching the sequence of the optional variadic parameters.

generates a metafunction called "trait" where 'trait' is the first macro parameter. template<class BOOST_TTI_TP_T> struct trait { static const value = unspecified; typedef mpl::bool_<true-or-false> type; }; The metafunction types and return: BOOST_TTI_TP_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.

Examples:

1) Search for an inner class template called 'MyTemplate', with all template type parameters, nested within the class 'MyClass' using a metafunction name of 'MyMeta'.

BOOST_TTI_TRAIT_HAS_TEMPLATE(MyMeta,MyTemplate)

or

BOOST_TTI_TRAIT_HAS_TEMPLATE(MyMeta,MyTemplate,BOOST_PP_NIL) // Non-variadic macro form

MyMeta<MyClass>::value

is a compile time boolean constant which is either 'true' or 'false' if the nested template exists.

2) Search for an inner class template called 'MyTemplate', with template parameters of 'class T,int x,template<class> class U', nested within the class 'MyClass' using a metafunction name of 'MyMeta'.

BOOST_TTI_TRAIT_HAS_TEMPLATE(MyMeta,MyTemplate,class,int,template<class> class)

or

BOOST_TTI_TRAIT_HAS_TEMPLATE(MyMeta,MyTemplate,(3,(class,int,template<class> class))) // Non-variadic macro form

MyMeta<MyClass>::value

is a compile time boolean constant which is either 'true' or 'false' if the nested template exists.

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(...)
```

Description

... = variadic parameters. The first variadic parameter is the inner class template name. Following variadic parameters are optional. If no following variadic parameters exist, then the inner class template being introspected must be all template type parameters (template parameters starting with `class` or `typename`) and any number of template type parameters can occur. If the second variadic parameter is BOOST_PP_NIL and no other variadic parameter is given, then just as in the previous case the inner class template being introspected must be all template type parameters (template parameters starting with `class` or `typename`) and any number of template type parameters can occur. This form is allowed in order to be consistent with using the non-variadic form of this macro. If the second variadic parameter is a Boost preprocessor library array and no other variadic parameter is given, then the

inner class template must have its template parameters matching the sequence in the tuple portion of the Boost PP array. This form is allowed in order to be consistent with using the non-variadic form of this macro. Otherwise the inner class template must have its template parameters matching the sequence of the optional variadic parameters.

generates a metafunction called "has_template_'name'" where 'name' is the first variadic parameter. template<class BOOST_TTI_TP_T> struct has_template_'name' { static const value = unspecified; typedef mpl::bool_<true-or-false> type; }; The metafunction types and return: BOOST_TTI_TP_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.

Examples:

1) Search for an inner class template called 'MyTemplate', with all template type parameters, nested within the class 'MyClass'.

BOOST_TTI_HAS_TEMPLATE(MyTemplate)

or

BOOST_TTI_HAS_TEMPLATE(MyTemplate,BOOST_PP_NIL) // Non-variadic macro form

has_template_MyTemplate<MyClass>::value

is a compile time boolean constant which is either 'true' or 'false' if the nested template exists.

2) Search for an inner class template called 'MyTemplate' with template parameters of 'class T,int x,template<class> class U' nested within the class 'MyClass'.

BOOST_TTI_HAS_TEMPLATE(MyTemplate,class,int,template<class> class)

or

BOOST_TTI_HAS_TEMPLATE(MyTemplate,(3,(class,int,template<class> class))) // Non-variadic macro form

has_template_MyTemplate<MyClass>::value

is a compile time boolean constant which is either 'true' or 'false' if the nested template exists.

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

Synopsis

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

BOOST_TTI_TRAIT_HAS_TYPE(trait, name)
```

Description

BOOST_TTI_TRAIT_HAS_TYPE is a macro which expands to a metafunction. The metafunction tests whether an inner type with a particular name exists and, optionally, whether a lambda expression invoked with the inner type is true or not.

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

name = the name of the inner type.

generates a metafunction called "trait" where 'trait' is the macro parameter. template<class BOOST_TTI_TP_T,class BOOST_TTI_TP_U> struct trait { static const value = unspecified; typedef mpl::bool_<true-or-false> type; }; The metafunction types and return: BOOST_TTI_TP_T = the enclosing type in which to look for our 'name'. BOOST_TTI_TP_U = (optional) An optional template parameter, defaulting to a marker type. If specified it is an MPL lambda expression which is invoked with the inner type found and must return a constant boolean value. returns = 'value' depends on whether or not the optional BOOST_TTI_TP_U is specified. If BOOST_TTI_TP_U is not specified, then 'value' is true if the 'name' type exists within the enclosing type BOOST_TTI_TP_T; otherwise 'value' is false. If BOOST_TTI_TP_U is specified , then 'value' is true if the 'name' type exists within the enclosing type BOOST_TTI_TP_T and the lambda expression as specified by BOOST_TTI_TP_U, invoked by passing the actual inner type of 'name', returns a 'value' of true; otherwise 'value' is false. The action taken with BOOST_TTI_TP_U occurs only when the 'name' type exists within the enclosing type BOOST_TTI_TP_T.

Example usage:

BOOST_TTI_TRAIT_HAS_TYPE(LookFor,MyType) generates the metafunction LookFor in the current scope to look for an inner type called MyType.

LookFor<EnclosingType>::value is true if MyType is an inner type of EnclosingType, otherwise false.

LookFor<EnclosingType,ALambdaExpression>::value is true if MyType is an inner type of EnclosingType and invoking ALambdaExpression with the inner type returns a value of true, otherwise false.

A popular use of the optional MPL lambda expression is to check whether the type found is the same as another type, when the type found is a typedef. In that case our example would be:

LookFor<EnclosingType,boost::is_same<_,SomeOtherType> >::value is true if MyType is an inner type of EnclosingType and is the same type as SomeOtherType.

Macro BOOST_TTI_HAS_TYPE

BOOST_TTI_HAS_TYPE

Synopsis

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

BOOST_TTI_HAS_TYPE(name)
```

Description

BOOST_TTI_HAS_TYPE is a macro which expands to a metafunction. The metafunction tests whether an inner type with a particular name exists and, optionally, whether a lambda expression invoked with the inner type is true or not.

name = the name of the inner type.

generates a metafunction called "has_type_'name'" where 'name' is the macro parameter. template<class BOOST_TTI_TP_T,class BOOST_TTI_TP_U> struct has_type_'name' { static const value = unspecified; typedef mpl::bool_<true-or-false> type; }; The metafunction types and return: BOOST_TTI_TP_T = the enclosing type in which to look for our 'name'. BOOST_TTI_TP_U = (optional) An optional template parameter, defaulting to a marker type. If specified it is an MPL lambda expression which is invoked with the inner type found and must return a constant boolean value. returns = 'value' depends on whether or not the optional BOOST_TTI_TP_U is specified. If BOOST_TTI_TP_U is not specified, then 'value' is true if the 'name' type exists within the enclosing type BOOST_TTI_TP_T; otherwise 'value' is false. If BOOST_TTI_TP_U is specified , then 'value' is true if the 'name' type exists within the enclosing type BOOST_TTI_TP_T and the lambda expression as specified by BOOST_TTI_TP_U, invoked by passing the actual inner type of 'name', returns a 'value' of true; otherwise 'value' is false. The action taken with BOOST_TTI_TP_U occurs only when the 'name' type exists within the enclosing type BOOST_TTI_TP_T.

Example usage:

BOOST_TTI_HAS_TYPE(MyType) generates the metafunction has_type_MyType in the current scope to look for an inner type called MyType.

has_type_MyType<EnclosingType>::value is true if MyType is an inner type of EnclosingType, otherwise false.

has_type_MyType<EnclosingType,ALambdaExpression>::value is true if MyType is an inner type of EnclosingType and invoking ALambdaExpression with the inner type returns a value of true, otherwise false.

A popular use of the optional MPL lambda expression is to check whether the type found is the same as another type, when the type found is a typedef. In that case our example would be:

has_type_MyType<EnclosingType,boost::is_same<_,SomeOtherType> >::value is true if MyType is an inner type of EnclosingType and is the same type as SomeOtherType.

Header <boost/tti/member_type.hpp>

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

```
namespace boost {
  namespace tti {
    template<typename BOOST_TTI_TP_T,
             typename BOOST_TTI_TP_MARKER_TYPE = BOOST_TTI_NAMESPACE::detail::notype>
      struct valid_member_type;
    template<typename TTI_METAFUNCTION> struct valid_member_metafunction;
  }
}
```

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 BOOST_TTI_MEMBER_TYPE (BOOST_TTI_TRAIT_MEMBER_TYPE) is a valid type.

Synopsis

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

template<typename BOOST_TTI_TP_T,
         typename BOOST_TTI_TP_MARKER_TYPE = BOOST_TTI_NAMESPACE::detail::notype>
struct valid_member_type : public boost::mpl::not_< boost::is_same< BOOST_TTI_TP_T, ↵
BOOST_TTI_TP_MARKER_TYPE > >
{
};
```

Description

template<class BOOST_TTI_TP_T,class BOOST_TTI_TP_MARKER_TYPE = boost::tti::detail::notype> struct valid_member_type { static const value = unspecified; typedef mpl::bool_<true-or-false> type; };

The metafunction types and return:

BOOST_TTI_TP_T = returned inner 'type' from invoking the macro metafunction generated by BOOST_TTI_MEMBER_TYPE (BOOST_TTI_TRAIT_MEMBER_TYPE). BOOST_TTI_TP_MARKER_TYPE = (optional) a type to use as the marker type. defaults to the internal boost::tti::detail::notype.

returns = 'value' is true if the type is valid, otherwise 'value' is false. A valid type means that the returned inner 'type' is not the marker type.

Struct template valid_member_metafunction

boost::tti::valid_member_metafunction — A metafunction which checks whether the invoked macro metafunction generated by BOOST_TTI_MEMBER_TYPE (BOOST_TTI_TRAIT_MEMBER_TYPE) hold a valid type.

Synopsis

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

template<typename TTI_METAFUNCTION>
struct valid_member_metafunction : public boost::mpl::not_< boost::is_same< TTI_METAFUNC↵
TION::type, TTI_METAFUNCTION::boost_tti_marker_type > >
{
};
```

Description

template<class TTI_METAFUNCTION> struct valid_member_metafunction { static const value = unspecified; typedef mpl::bool_<true-or-false> type; };

The metafunction types and return:

TTI_METAFUNCTION = The invoked macro metafunction generated by BOOST_TTI_MEMBER_TYPE (BOOST_TTI_TRAIT_MEMBER_TYPE).

returns = 'value' is true if the nested type of the invoked metafunction is valid, otherwise 'value' is false. A valid type means that the invoked metafunction's inner 'type' is not the marker type.

Macro BOOST_TTI_TRAIT_MEMBER_TYPE

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

generates a metafunction called "trait" where 'trait' is the macro parameter. template<class BOOST_TTI_TP_T,class BOOST_TTI_TP_MARKER_TYPE = boost::tti::detail::notype> struct trait { typedef unspecified type; typedef BOOST_TTI_TP_MARKER_TYPE boost_tti_marker_type; }; The metafunction types and return: BOOST_TTI_TP_T = the enclosing type. BOOST_TTI_TP_MARKER_TYPE = (optional) a type to use as the marker type. defaults to the internal boost::tti::detail::notype.

returns = 'type' is the inner type of 'name' if the inner type exists within the enclosing type, else 'type' is a marker type. if the end-user does not specify a marker type then an internal boost::tti::detail::notype marker type is used. The metafunction also encapsulates the type of the marker type as a nested 'boost_tti_marker_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 an evaluation of inner type existence, without generating a compiler error, 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 a marker type.

Synopsis

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

BOOST_TTI_MEMBER_TYPE(name)
```

Description

name = the name of the inner type.

generates a metafunction called "member_type_name" where 'name' is the macro parameter. template<class BOOST_TTI_TP_T,class BOOST_TTI_TP_MARKER_TYPE = boost::tti::detail::notype> struct member_type_name { typedef unspecified type; typedef BOOST_TTI_TP_MARKER_TYPE boost_tti_marker_type; }; The metafunction types and return: BOOST_TTI_TP_T = the enclosing type. BOOST_TTI_TP_MARKER_TYPE = (optional) a type to use as the marker type. defaults to the internal boost::tti::detail::notype. returns = 'type' is the inner type of 'name' if the inner type exists within the enclosing type, else 'type' is a marker type. if the end-user does not specify a marker type then an internal boost::tti::detail::notype marker type is used. The metafunction also encapsulates the type of the marker type as a nested 'boost_tti_marker_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 an evaluation of inner type existence, without generating a compiler error, which can be used by other metafunctions in this 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, or specifying the target 'tti', will run tests for both basic TTI and for the variadic macro portion of TTI. You can run tests for only the basic TTI, which is the vast majority of TTI functionality, by specifying only the 'ttinovm' target when executing the jamfile. 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 successfully tested with:

- VC++ 8, 9, 10, 11
- gcc 3.4.2, 3.4.5, 4.3.0, 4.4.0, 4.4.7, 4.5.0-1, 4.5.2-1, 4.6.0, 4.6.1, 4.6.2, 4.6.3, 4.7.0, and 4.7.2.
- clang 2.8, 3.0, 3.1, and 3.3 (latest)
- Intel-linux 12.1, 13.0, 13.1

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