Chapter . The Type Traits Introspection Library 1.0

Edward Diener

Table of Contents

Introduction

Welcome to the Type Traits Introspection library version 1.1 .

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

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

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

The functionality of the library may be summed up as:

- Provide the means to introspect a type at compile time using a set of macros. Each macro takes the name of the type's element and generates a metafunction which can be subsequently invoked to determine whether or not the element exists within the type. These metafunctions will be called "macro metafunctions" in the documentation.

- Provide corresponding metafunctions which can operate on nearly each of the macro metafunctions generated. These secondary metafunctions provide nearly the same set of introspection as the macro metafunctions but allow for an easier to use syntax, where the types being passed are in the form of nullary metafunctions themselves. They always work with individual types when specifying

function and data syntax rather than a composite type. These metafunctions are called 'nullary type metafunctions' in the documentation.

The library is a header only library.

There are two separate headers in the library, divided depending on whether or not the library functionality supporting variadic macros is to be used.

1. The main header, which does not require using the library support for variadic macros, is `'TTIntrospection.hpp'`. This can be used for the vast majority of functionality in the library.

2. The secondary header, which uses a very small subset of the library functionality, providing an alternate use of a particular macro with variadic macro support, is `'TTIntrospectionVM.hpp'`.

Furthermore there is a third header file which both of the header files above include, called 'TTIntrospectionTemplate.hpp' but this should never be included itself but only through either of the main header files.

The library is dependent on Boost PP, Boost MPL, Boost Type Traits, and Boost Function Types.

If the secondary header is used the library is also dependent on the variadic_macro_data library currently in the sandbox. If the secondary header is not used there is no need to download the variadic_macro_data library and use it in any way.

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

Why the TTI Library ?

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

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

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

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

Terminology

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

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

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

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

General Functionality

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

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

These are the compile-time questions which the TTI library answers.

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 TTI_, create their metafunctions in a top-level namespace called 'tti', and come in two forms:

1. In the simplest 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. To generate the name of the metafunction the 'name' is appended to the name of the macro, with the TTI_ prefix removed, a final underscore added, and the macro part of the name in lower case. As an example, for the macro TTI_HAS_TYPE(MyType) the name of the metafunction is 'tti::has_type_MyType' and it will look for an inner type called 'MyType'.

2. In the slightly more complicated form, which I call the complex form, the macro starts with 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 only to completely name the metafunction in the tti namespace. As an example, for the macro TTI_TRAIT_HAS_TYPE(MyTrait,MyType) the name of the metafunction is 'tti::MyTrait' and it will look for an inner type called 'MyType'. Every macro has a corresponding complex form.

Important

When introspecting a particular inner element any given macro metafunction generated can be reused with any combination of template parameters which involve the same type of inner element. Furthermore once a macro metafunction is generated, attempting to generate another macro metafunction of the same name will create ODR violations since two C++ constructs with the same name/type in the same namespace will have been created. This latter possibility has much less chance of occurence if you use the simple form of each macro and just reuse the macro metafunction. You can even do this if you are introspecting for two entities of the same name in different enclosing types, or in the same enclosing type but with different signatures, as with overloaded member functions.

Once either of these two macro forms are used for a particular type of inner element, the corresponding macro metafunction 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 name can always alternatively be used.

Macro Metafunctions

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

To use these metafunctions you need to include the main header file 'TTIntrospection.hpp', unless otherwise noted.

A table of these macros is given, with the inner element whose existence the metaprogrammer is introspecting. A more detailed explanation can be found in the reference section, and examples of usage 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 tti::trait where 'trait' is the first parameter to the corresponding complex form macro. All of the introspecting metafunctions in the table below return a boolean constant called 'value', which specifies whether or not the inner element exists.

Table 1. TTI Macro Metafunctions

Inner Element	Macro	Template
Type	`TTI_HAS_TYPE`(name)	tti::has_type_'name' class T = enclosing type
Type with check	`TTI_HAS_TYPE`(name)	tti::has_type_'name' class T = enclosing type class U = type to check against
Class Template	`TTI_HAS_TEMPLATE`(name)	tti::has_template_'name' class T = enclosing type All of the template parameters must be 'class' (or 'typename') parameters
Class Template with params	`TTI_HAS_TEMPLATE_CHECK_PARAMS`(name,ppSeq[a])	tti::has_template_check_params_'name' class T = enclosing type
Class Template with params using variadic macros[b]	`TTI_VM_HAS_TEMPLATE_CHECK_PARAMS`(name,...[c])	tti::has_template_check_params_'name' class T = enclosing type
Member data or function	`TTI_HAS_MEMBER`(name)	tti::has_member_'name' class T = pointer to data or function member The form for T is 'Type Class::*' for member data The form for T is 'ReturnType (Class::*)(Zero or more comma-separated parameter types)' for member function
Member data as individual types	`TTI_HAS_MEMBER_DATA`(name)	tti::has_member_data_'name' class T = enclosing type class R = data type

Inner Element	Macro	Template
Member function as individual types	TTI_HAS_MEMBER_FUNCTION(name)	tti::has_member_function_'name' class T = enclosing type class R = return type class FS = (optional) function parameter types as a Boost MPL forward sequence. If there are no function parameters this does not have to be specified. Defaults to boost::mpl::vector<>. class TAG = (optional) Boost function_types tag type. Defaults to boost::function_types::null_tag.
Static member data or static member function	TTI_HAS_STATIC_MEMBER(name)	tti::has_static_member_'name' class T = enclosing type class Type = data or function type The form for Type is just 'Type' as a data type The form for Type is 'ReturnType (Zero or more comma-separated parameter types)' as a function type
Static member function as individual types	TTI_HAS_STATIC_MEMBER_FUNCTION(name)	tti::has_static_member_function_'name' class T = enclosing type class R = return type class FS = (optional) function parameter types as a Boost MPL forward sequence. If there are no function parameters this does not have to be specified. Defaults to boost::mpl::vector<>. class TAG = (optional) Boost function_types tag type. Defaults to boost::function_types::null_tag.

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

[b] Header file is TTIntrospectionVM.hpp.

[c] The template parameters as variadic data.

There is one other macro which creates a metafunction which does not introspect for the existence of an inner element type, but is very useful nonetheless. Instead the macro metafunction created returns the nested type if it exists, else it returns an unspecified type.

Table 2. TTI Nested Type Macro Metafunction

Inner Element	Macro	Template
Type	`TTI_MEMBER_TYPE`(name)	tti::member_type_'name' class T = enclosing type returns = the type of 'name' if it exists, else an unspecified type, as a typedef 'type'.

Along with this macro metafunction we have another metafunction which, when passed a 'type', which can be any type but which will generally be used with the type returned by invoking the metafunction generated by TTI_MEMBER_TYPE, will tell use whether the type exists or not as a boolean constant.

Table 3. TTI Nested Type Macro Metafunction Existence

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

The usefulness of the TTI_MEMBER_TYPE macro metafunction will be shown in the next topic when I explain the problem of specifying nested types and how TTI solves it.

Nested Types

The problem

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

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

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

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 TTI_HAS_TYPE(InnerType) macro and it's related tti::has_type_InnerType metafunction to determine if the nested type 'InnerType' exists. This solution is perfectly valid and, with Boost MPL's selection metafunctions, we can do compile-time selection to generate the correct template code.

However this does not 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 TTI library offers a better solution in the form of constructs which work with nested types without producing a compiler error if the nested type does not exist, but still are able to do the introspecting for inner elements that our TTI macro metafunctions do.

We have already seen one of those constructs, the macro TTI_MEMBER_TYPE, which generates a metafunction based on the name of an inner type. But instead of telling us whether that inner type exists it instead returns a typedef 'type' which is that inner type if it exists, else it is an unspecified type if it does not. In this way we have created a metafunction, very similar in functionality to boost::mpl::identity, but which still returns some unspecified marker 'type' if our nested type is invalid.

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

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

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

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

```
TTI_MEMBER_TYPE(FindType)
TTI_MEMBER_TYPE(AType)
TTI_MEMBER_TYPE(BType)
TTI_MEMBER_TYPE(CType)
```

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

```
typedef typename
tti::member_type_FindType
  <
  typename tti::member_type_CType
    <
    typename tti::member_type_BType
      <
      typename tti::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.

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

```
TTI_HAS_STATIC_MEMBER(MyData)
```

Next we create our metafunction:

```
tti::has_static_member_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.

We can also directly find out whether the deeply nested type 'FindType' actually exists in a similar manner. Our metafunction would be:

```
TTI_HAS_TYPE(FindType)

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

Because this duplicates much of our code for the 'MyFindType' typedef to create our nested type, we can instead, and much more easily, pass our type 'MyFindType', since we already have it in the form of a type, to another metafunction called 'tti::valid_member_type', which returns a boolean constant which is 'true' if our nested exists or 'false' if it does not.

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

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

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

The using of TTI_MEMBER_TYPE to create a nested type which may or may not exist, and which can subsequently be used with our macro metafunctions whenever a nested type is required, without producing a compiler error when the type does not actually exist, is the main reason we have separate but similar functionality among our macro metafunctions to determine whether a member data, a member function, or a static member function exists within an enclosing type.

In the more general case, when using TTI_HAS_MEMBER and TTI_HAS_STATIC_MEMBER, the signature for the member data, member function, and the function portion of a static member function is a composite type. This makes for a syntactical notation which is easy to specify, but because of that composite type notation we can not use the nested type functionality in TTI_MEMBER_TYPE very easily. But when we use the TTI_HAS_MEMBER_DATA, TTI_HAS_MEMBER_FUNCTION, and TTI_HAS_STATIC_MEMBER_FUNCTION the composite types in our signatures are broken down into their individual types so that using TTI_MEMBER_TYPE, if necessary, for one of the individual types is easy.

A more elegant solution

Although using TTI_MEMBER_TYPE represents a good solution to creating a nested type without the possible compile-time error of the T::InnerType syntax, reaching in to specify all those ::type expressions, along with their repeated 'typename', does get syntactically tedious.

Because of this the TTI library offers a parallel set of metafunctions to the macro metafunctions where the 'types' specified are themselves nullary metafunctions. This parallel set of metafunctions, using nullary metafunctions to specify individual types, rather than the actual types themselves, are called 'nullary type metafunctions'. In this group there is also a nullary metafunction paralleling our TTI_MEMBER_TYPE macro metafunction, and therefore a further construct making the specifying of nested types easy and error-free to use.

This group of nullary type metafunctions will be fully explained later after we give some examples of macro metafunction use.

Using the Macro Metafunctions

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

```
// An enclosing type

struct AType
  {

  // Type

  typedef int AnIntType; // as a typedef

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

  // Template

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

  // Data

  BType IntBT;

  // Function

  int IntFunction(short) { return 0; }

  // Static Data

  static short DSMember;

  // Static Function

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

  };
```

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

```
#include <boost/TTIntrospection.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' ?

```
TTI_HAS_TYPE(AnIntType)

tti::has_type_AnIntType
  <
  T
  >
```

Does T have a nested type called 'BType' ?

```
TTI_HAS_TYPE(BType)

tti::has_type_BType
  <
  T
  >
```

Type checking the typedef

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

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

Template

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

```
TTI_HAS_TEMPLATE(AMemberTemplate)

tti::has_template_AMemberTemplate
  <
  T
  >
```

Template with params

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

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

tti::has_template_check_params_MoreParameters
  <
  T
  >
```

Template with params using variadic macros

Note

Include the 'TTIntrospectionVM.hpp' header file when using this macro.

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

```
TTI_VM_HAS_TEMPLATE_CHECK_PARAMS(ManyParameters,class,class,int,class,template <class> class In↵
nerTemplate,class,long)

tti::has_template_check_params_ManyParameters
  <
  T
  >
```

Member data with composite type

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

```
TTI_HAS_MEMBER(IntBT)

tti::has_member_IntBT
  <
  AType::BType T::*
  >
```

Member data with individual types

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

```
TTI_HAS_MEMBER_DATA(IntBT)

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

Member function with composite type

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

```
TTI_HAS_MEMBER(IntFunction)

tti::has_member_IntFunction
  <
  int (T::*)(short)
  >
```

Member function with individual types

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

```
TTI_HAS_MEMBER_FUNCTION(IntFunction)

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

Static member data

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

```
TTI_HAS_STATIC_MEMBER(DSMember)

tti::has_static_member_DSMember
  <
  T,
  short
  >
```

Static member function with composite type

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

```
TTI_HAS_STATIC_MEMBER(SIntFunction)

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

Static member function with individual types

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

```
TTI_HAS_STATIC_MEMBER(SIntFunction)

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

Member type

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

```
TTI_MEMBER_TYPE(BType)
TTI_MEMBER_TYPE(CType)

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

Member type existence

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

```
TTI_MEMBER_TYPE(BType)
TTI_MEMBER_TYPE(CType)

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

tti::valid_member_type
  <
  AType
  >
```

Nullary Type Metafunctions

The nullary type metafunctions parallel most of the macro metafunctions but more easily allow a syntax where nested types can be specified without needing to manually reach into the 'type' member of TTI_MEMBER_TYPE or its nullary type metafunction equivalent of tti::mf_member_type. In a very real way the nullary type metafunctions exist just to provide syntactic improvements over the macro metafunctions and are not needed to use the library, since all of the library functionality is already provided with the macro metafunctions. Nonetheless syntactic ease of use is a very real goal of the TTI library and therefore these metafunctions are provided to allow that syntactic improvement.

For each of these nullary type metafunctions the first parameter is a Boost MPL lambda expression using its corresponding the macro metafunction to pass metafunctions as data. The easiest way to do this is to use a Boost MPL placeholder expression. The syntax for passing the corresponding macro metafunction becomes 'macrometafunction<>' etc. depending on how many parameters are bring passed. Thus for two parameters we would have 'macrometafunction<,>' etc., with another placeholder (') added for each subsequent parameter.

The remaining parameter are 'types'. These 'types' always consist first of the enclosing type and then possibly other types which make up the signature of whatever inner element we are introspecting. Each of these 'types' is passed as a nullary metafunction whose typedef 'type' is the actual type.

The only exception to this use of nullary type metafunctions when specifying 'types' is when a Boost function_types tag type, which is optional, is specified as an addition to the function signature. Also when dealing with a function signature and parameter types

are being passed, while the parameter 'types' themselves are in the form of nullary metafunctions, the MPL forward sequence which contains the parameter 'types' should not be wrapped as a nullary metafunction.

For a type which is in scope, we can always use boost::mpl::identity to create our nullary metafunction, and there can never be a compiler error for such known types as long as declarations for them exist or it is a built-in C++ type. For nested types, which may or may not exist, we can pass the result of TTI_MEMBER_TYPE or its equivalent nullary type metafunction tti::mf_member_type (explained later).

To use these metafunctions you need to include the main header file 'TTIntrospection.hpp', unless otherwise noted.

Tip

The header file <boost/mpl/identity.hpp> is included by the TTI header files so you need not manually include it in order to use boost::mpl::identity to wrap a known type as a nullary metafunction for the nullary type metafunctions. Also the header file <boost/mpl/vector.hpp> is included by the main TTI header file 'TTIntrospection.hpp' so if you use an MPL vector as your forward sequence wrapper for parameter types, you need not manually include the header file. Finally the header file <boost/mpl/placeholders.hpp> is also included by 'TTIntrospection.hpp' so you need not manually include it yourself in order to use placeholder expressions.

A table of these metafunctions is given, based on the inner element whose existence the metaprogrammer is introspecting. A more detailed explanation can be found in the reference section, and examples of usage can be found in the "Using the Nullary Type Metafunctions" section. All of the metafunctions are in the top-level 'tti' namespace, all have a particular name based on the type of its functionality, and all begin with the prefix 'mf_' so as not to conflict with the macro metafunction names generated by the library.

Table 4. TTI Nullary Type Metafunctions

Inner Element	Template	Parameters	Macro Equivalent
Type	`tti::mf_has_type`	class HasType = macro metafunction as lambda expression class T = enclosing type nullary metafunction	TTI_HAS_TYPE
Type with check	`tti::mf_has_type`	class HasType = macro metafunction as lambda expression class T = enclosing type nullary metafunction class U = type to check against nullary metafunction	TTI_HAS_TYPE
Class Template	`tti::mf_has_template`	class HasTemplate = macro metafunction as lambda expression class T = enclosing type nullary metafunction	TTI_HAS_TEMPLATE
Class Template with params	`tti::mf_has_template_check_params`	class HasTemplateCheckParams = macro metafunction as lambda expression class T = enclosing type nullary metafunction	TTI_HAS_TEMPLATE_CHECK_PARAMS TTI_VM_HAS_TEMPLATE_CHECK_PARAMS
Member data	`tti::mf_has_member_data`	class HasMemberData = macro metafunction as lambda expression class T = enclosing type nullary metafunction class R = type of member data nullary Metafunction	TTI_HAS_MEMBER_DATA

Inner Element	Template	Parameters	Macro Equivalent
Member function	`tti::mf_has_member_function`	class HasMemberFunction = macro metafunction as lambda expression class T = enclosing type nullary metafunction class R = return value nullary metafunction class FS = (optional) a Boost MPL forward sequence of parameter types as nullary metafunctions. The forward sequence as a type is not presented as a nullary metafunction. If there are no parameters, this may be omitted. class TAG = (optional) a Boost function_types tag type.	TTI_HAS_MEMBER_FUNCTION
Static data	`tti::mf_has_static_data`	class HasStaticMember = macro metafunction as lambda expression class T = enclosing type nullary metafunction class R = type of static data nullary metafunction	TTI_HAS_STATIC_MEMBER
Static function	`tti::mf_has_static_member_function`	class HasStaticMemberFunction = macro metafunction as lambda expression class T = enclosing type nullary metafunction class R = return value nullary metafunction class FS = (optional) a Boost MPL forward sequence of parameter types as nullary metafunctions. The forward sequence as a type is not presented as a nullary metafunction. If there are no parameters, this may be omitted. class TAG = (optional) a Boost function_types tag type.	TTI_HAS_STATIC_MEMBER_FUNCTION

Other than the use of nearly all types as nullary metafunctions, one other difference in the nullary type metafunctions from their macro metafunction counterparts is that the signature for member functions, member data, and static member functions always involves individual types rather than the combined type notation which some of the macro metafunctions use. This allows us to specify nested types in those signatures without using the T::InnerType notation.

Nullary type metafunction member_type equivalent

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

Table 5. TTI Nested Type Nullary Type Metafunction

Inner Element	Template	Parameters	Macro Equivalent
Type	`tti::mf_member_type`	class MemberType = macro metafunction as lambda expression class T = enclosing type nullary metafunction	TTI_MEMBER_TYPE

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

The use of this metafunction allows us to created deeply nested types, which may or may not exist, as nullary metafunctions in much the same way that TTI_MEMBER_TYPE can. The difference is the simpler syntax when using tti::mf_member_type.

As an example, given the theoretical relationship of types we used before:

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

We can use tti::mf_member_type as follows. First we create our corresponding macro metafunctions:

```
TTI_MEMBER_TYPE(FindType)
TTI_MEMBER_TYPE(AType)
TTI_MEMBER_TYPE(BType)
TTI_MEMBER_TYPE(CType)
```

Next we can create a typedef to reflect a nested type called FindType, as a nullary metafunction, which has the relationship as specified above by using tti::mf_member_type.

```
typedef
tti::mf_member_type
  <
  tti::member_type_FindType<_>,
  tti::mf_member_type
    <
    tti::member_type_CType<_>,
    tti::mf_member_type
      <
      tti::member_type_BType<_>,
      tti::member_type_AType
        <
        T
        >
      >
    >
  > MyFindType;
```

The nested type created can be used with the other nullary type metafunctions above. The key information above is that the enclosing type, as in all of the nullary type metafunctions, is a nullary metafunction itself, which means that the enclosing type can be specified as the result of using TTI_MEMBER_TYPE as well as the result of using mf_member_type itself.

Both techniques are shown in the example above, and the same technique for creating nested types as nullary metafunctions can be used with the other functionality of the nullary type metafunctions when nested types are needed as 'types'.

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

Table 6. TTI Nested Type Nullary Type Metafunction Existence

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

Again note the difference here from the equivalent macro metafunction tti::valid_member_type. In the example above the type T is passed as a nullary metafunction holding the actual type, where for the macro metafunction equivalent the type T is passed as the actual type being tested.

In our next section we will look at examples of nullary type metafunction use.

Using the Nullary Type Metafunctions

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

```
// An enclosing type

struct T
  {

  // Type

  struct BType // as a nested type
    {

    // Template

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

    // Type

    struct CType // as a further nested type
      {

      // Template

      template <class> struct AMemberTemplate { };

      // Type

      struct DType // as a futher nested type
        {

        // Type

        typedef double ADoubleType;

        // Template

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

        // Function

        int IntFunction(short) const { return 0; }

        // Static Data

        static short DSMember;

        // Static Function

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

        };
      };
    };

  // Data

  BType IntBT;

  };
```

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

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

template<class T>
struct OurTemplateClass
  {

  // compile-time template code regarding T

  };
```

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

Member type

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

```
TTI_MEMBER_TYPE(BType)
TTI_MEMBER_TYPE(CType)
TTI_MEMBER_TYPE(DType)

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

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

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

We will use these typedefs in the ensuing examples.

Type

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

```
TTI_HAS_TYPE(DType)

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

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

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

Type checking the typedef

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

```
TTI_HAS_TYPE(ADoubleType)

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

Template

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

```
TTI_HAS_TEMPLATE(AMemberTemplate)

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

Template with params

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

```
TTI_HAS_TEMPLATE_CHECK_PARAMS(ManyParameters,(class)(class)(int)(class)(template <class> class In↵
nerTemplate)(class)(long))

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

Template with params using variadic macros

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

Note

Include the 'TTIntrospectionVM.hpp' header file when using this macro.

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

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

Member data

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

```
TTI_HAS_MEMBER_DATA(IntBT)

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

Member function

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

```
TTI_HAS_MEMBER_FUNCTION(IntFunction)

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

Static member data

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

```
TTI_HAS_STATIC_MEMBER(DSMember)

tti::mf_has_static_data
  <
  tti::has_static_member_DSMember<_,_>,
  DTypeNM,
  boost::mpl::identity<short>
  >
```

Static member function

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

```
TTI_HAS_STATIC_MEMBER_FUNCTION(SIntFunction)

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

TypeTraitsIntrospection Reference

Header <boost/tti/TTIntrospection.hpp>

```
TTI_TRAIT_HAS_TYPE(trait, name)
TTI_HAS_TYPE(name)
TTI_TRAIT_MEMBER_TYPE(trait, name)
TTI_MEMBER_TYPE(name)
TTI_TRAIT_HAS_TEMPLATE(trait, name)
TTI_HAS_TEMPLATE(name)
TTI_TRAIT_HAS_TEMPLATE_CHECK_PARAMS(trait, name, tpSeq)
TTI_HAS_TEMPLATE_CHECK_PARAMS(name, tpSeq)
TTI_TRAIT_HAS_MEMBER(trait, name)
TTI_HAS_MEMBER(name)
TTI_TRAIT_HAS_MEMBER_FUNCTION(trait, name)
TTI_HAS_MEMBER_FUNCTION(name)
TTI_TRAIT_HAS_MEMBER_DATA(trait, name)
TTI_HAS_MEMBER_DATA(name)
TTI_TRAIT_HAS_STATIC_MEMBER(trait, name)
TTI_HAS_STATIC_MEMBER(name)
TTI_TRAIT_HAS_STATIC_MEMBER_FUNCTION(trait, name)
TTI_HAS_STATIC_MEMBER_FUNCTION(name)
```

```
namespace tti {
  template<typename T> struct valid_member_type;
  template<typename T> struct mf_valid_member_type;
  template<typename HasType, typename T,
           typename U = boost::mpl::identity<tti::detail::notype> >
    struct mf_has_type;
  template<typename MemberType, typename T> struct mf_member_type;
  template<typename HasTemplate, typename T> struct mf_has_template;
  template<typename HasMemberFunction, typename T, typename R,
           typename FS = boost::mpl::vector<>,
           typename TAG = boost::function_types::null_tag>
    struct mf_has_member_function;
  template<typename HasMemberData, typename T, typename R>
    struct mf_has_member_data;
  template<typename HasStaticMemberFunction, typename T, typename R,
           typename FS = boost::mpl::vector<>,
           typename TAG = boost::function_types::null_tag>
    struct mf_has_static_member_function;
  template<typename HasStaticMember, typename T, typename R>
    struct mf_has_static_data;
}
```

Struct template valid_member_type

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

Synopsis

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

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

Description

The metafunction types and return:

T = returned inner 'type' from invoking the macro metafunction generated by TTI_MEMBER_TYPE (TTI_TRAIT_MEMBER_TYPE) or from invoking tti::mf_member_type.

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

Struct template mf_valid_member_type

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

Synopsis

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

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

Description

The metafunction types and return:

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

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

Struct template mf_has_type

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

Synopsis

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

template<typename HasType, typename T,
         typename U = boost::mpl::identity<tti::detail::notype> >
struct mf_has_type {
};
```

Description

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

The metafunction types and return:

HasType = a Boost MPL lambda expression using the metafunction generated from the TTI_HAS_TYPE (or TTI_TRAIT_HAS_TYPE) macro.
The easiest way to generate the lambda expression is to use a Boost MPL placeholder expression of the form 'metafunction<_>' (or optionally 'metafunction<_,_>').
T = the enclosing type as a nullary metafunction.
U = the type of the inner type as a nullary metafunction, as an optional parameter.

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

Struct template mf_member_type

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

Synopsis

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

template<typename MemberType, typename T>
struct mf_member_type {
};
```

Description

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

The metafunction types and return:

MemberType = a Boost MPL lambda expression using the metafunction generated from the TTI_MEMBER_TYPE (or TTI_TRAIT_MEMBER_TYPE) macro.
The easiest way to generate the lambda expression is to use a Boost MPL placeholder expression of the form 'metafunction<_>'.
T = the enclosing type as a nullary metafunction.

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

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

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

Struct template mf_has_template

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

Synopsis

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

template<typename HasTemplate, typename T>
struct mf_has_template {
};
```

Description

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

The metafunction types and return:

HasTemplate = a Boost MPL lambda expression using the metafunction generated from the TTI_HAS_TEMPLATE (TTI_TRAIT_HAS_TEMPLATE) macro.
The easiest way to generate the lambda expression is to use a Boost MPL placeholder expression of the form 'metafunction<_>'.
T = the enclosing type as a nullary metafunction.

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

Struct template mf_has_member_function

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

Synopsis

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

template<typename HasMemberFunction, typename T, typename R,
         typename FS = boost::mpl::vector<>,
         typename TAG = boost::function_types::null_tag>
struct mf_has_member_function {
};
```

Description

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

The metafunction types and return:

HasMemberFunction = a Boost MPL lambda expression using the metafunction generated from the TTI_HAS_MEMBER_FUNCTION (or TTI_TRAIT_HAS_MEMBER_FUNCTION) macro.
The easiest way to generate the lambda expression is to use a Boost MPL placeholder expression of the form 'metafunction<_,_> (or optionally 'metafunction<_,_,_> or ' 'metafunction<_,_,_,_>)'.
T = the enclosing type as a nullary metafunction.
R = the return type of the member function as a nullary metafunction.
FS = an optional parameter which is the parameters of the member function, each as a nullary metafunction, as a boost::mpl forward sequence.
This parameter defaults to boost::mpl::vector<>.
TAG = an optional parameter which is a boost::function_types tag to apply to the member function.
This parameter defaults to boost::function_types::null_tag.

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

Struct template mf_has_member_data

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

Synopsis

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

template<typename HasMemberData, typename T, typename R>
struct mf_has_member_data {
};
```

Description

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

The metafunction types and return:

HasMemberData = a Boost MPL lambda expression using the metafunction generated from the TTI_HAS_MEMBER_DATA (or TTI_TRAIT_HAS_MEMBER_DATA) macro.
The easiest way to generate the lambda expression is to use a Boost MPL placeholder expression of the form 'metafunction<_,_>'.
T = the enclosing type as a nullary metafunction.
R = the type of the member data as a nullary metafunction.

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

Struct template mf_has_static_member_function

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

Synopsis

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

template<typename HasStaticMemberFunction, typename T, typename R,
         typename FS = boost::mpl::vector<>,
         typename TAG = boost::function_types::null_tag>
struct mf_has_static_member_function {
};
```

Description

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

The metafunction types and return:

HasStaticMemberFunction = a Boost MPL lambda expression using the metafunction generated from the TTI_HAS_STATIC_MEMBER_FUNCTION (or TTI_TRAIT_HAS_STATIC_MEMBER_FUNCTION) macro.
The easiest way to generate the lambda expression is to use a Boost MPL placeholder expression of the form 'metafunction<_,_> (or optionally 'metafunction<_,_,_> or ' 'metafunction<_,_,_,_>)'.
T = the enclosing type as a nullary metafunction.
R = the return type of the static member function as a nullary metafunction.
FS = an optional parameter which is the parameters of the static member function, each as a nullary metafunction, as a boost::mpl forward sequence.
TAG = an optional parameter which is a boost::function_types tag to apply to the static member function.

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

Struct template mf_has_static_data

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

Synopsis

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

template<typename HasStaticMember, typename T, typename R>
struct mf_has_static_data {
};
```

Description

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

The metafunction types and return:

HasStaticMember = a Boost MPL lambda expression using the metafunction generated from the TTI_HAS_STATIC_MEMBER (or TTI_TRAIT_HAS_STATIC_MEMBER) macro.
The easiest way to generate the lambda expression is to use a Boost MPL placeholder expression of the form 'metafunction<_,_>'.
T = the enclosing type as a nullary metafunction.
R = the type of the static member data as a nullary metafunction.

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

Macro TTI_TRAIT_HAS_TYPE

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

Synopsis

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

TTI_TRAIT_HAS_TYPE(trait, name)
```

Description

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

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

The metafunction types and return:

T = the enclosing type in which to look for our 'name'.
U = the type of the inner type named 'name' as an optional parameter.
returns = 'value' is true if the 'name' type exists within the enclosing type and, if type U is specified, the 'name' type is the same as the type U, otherwise 'value' is false.

Macro TTI_HAS_TYPE

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

Synopsis

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

TTI_HAS_TYPE(name)
```

Description

name = the name of the inner type.

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

The metafunction types and return:

T = the enclosing type in which to look for our 'name'.
U = the type of the inner type named 'name' as an optional parameter.
returns = 'value' is true if the 'name' type exists within the enclosing type and, if type U is specified, the 'name' type is the same as the type U, otherwise 'value' is false.

Macro TTI_TRAIT_MEMBER_TYPE

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

Synopsis

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

TTI_TRAIT_MEMBER_TYPE(trait, name)
```

Description

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

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

The metafunction types and return:

T = the enclosing type.
returns = 'type' is the inner type of 'name' if the inner type exists within the enclosing type, else 'type' is an unspecified type.

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

Macro TTI_MEMBER_TYPE

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

Synopsis

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

TTI_MEMBER_TYPE(name)
```

Description

name = the name of the inner type.

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

The metafunction types and return:

T = the enclosing type.
returns = 'type' is the inner type of 'name' if the inner type exists within the enclosing type, else 'type' is an unspecified type.

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

Macro TTI_TRAIT_HAS_TEMPLATE

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/TTIntrospection.hpp>

TTI_TRAIT_HAS_TEMPLATE(trait, name)
```

Description

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

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

The metafunction types and return:

T = the enclosing type in which to look for our 'name'.
returns = 'value' is true if the 'name' template exists within the enclosing type, otherwise 'value' is false.

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

Macro TTI_HAS_TEMPLATE

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

Synopsis

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

TTI_HAS_TEMPLATE(name)
```

Description

name = the name of the inner template.

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

The metafunction types and return:

T = the enclosing type in which to look for our 'name'.
returns = 'value' is true if the 'name' template exists within the enclosing type, otherwise 'value' is false.

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

Macro TTI_TRAIT_HAS_TEMPLATE_CHECK_PARAMS

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

Synopsis

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

TTI_TRAIT_HAS_TEMPLATE_CHECK_PARAMS(trait, name, tpSeq)
```

Description

trait = the name of the metafunction within the tti namespace.
name = the name of the inner class template.
tpSeq = a Boost PP sequence which has the class template parameters. Each part of the template parameters separated by a comma (,) is put in a separate sequence element.

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

The metafunction types and return:

T = the enclosing type in which to look for our 'name'.
returns = 'value' is true if the 'name' class template with the signature as defined by the 'tpSeq' exists within the enclosing type, otherwise 'value' is false.

Macro TTI_HAS_TEMPLATE_CHECK_PARAMS

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

Synopsis

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

TTI_HAS_TEMPLATE_CHECK_PARAMS(name, tpSeq)
```

Description

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

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

The metafunction types and return:

T = the enclosing type in which to look for our 'name'.
returns = 'value' is true if the 'name' class template with the signature as defined by the 'tpSeq' exists within the enclosing type, otherwise 'value' is false.

Macro TTI_TRAIT_HAS_MEMBER

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

Synopsis

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

TTI_TRAIT_HAS_MEMBER(trait, name)
```

Description

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

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

The metafunction types and return:

T = the type, in the form of a member data pointer or member function pointer, in which to look for our 'name'.
returns = 'value' is true if the 'name' exists, with the appropriate type, otherwise 'value' is false.

Macro TTI_HAS_MEMBER

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

Synopsis

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

TTI_HAS_MEMBER(name)
```

Description

name = the name of the inner member.

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

The metafunction types and return:

T = the type, in the form of a member data pointer or member function pointer, in which to look for our 'name'.
returns = 'value' is true if the 'name' exists, with the appropriate type, otherwise 'value' is false.

Macro TTI_TRAIT_HAS_MEMBER_FUNCTION

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/TTIntrospection.hpp>

TTI_TRAIT_HAS_MEMBER_FUNCTION(trait, name)
```

Description

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

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

The metafunction types and return:

T = the enclosing type in which to look for our 'name'.
R = the return type of the member function.
FS = an optional parameter which are the parameters of the member function as a boost::mpl forward sequence.
TAG = an optional parameter which is a boost::function_types tag to apply to the member function.
returns = 'value' is true if the 'name' exists, with the appropriate type, otherwise 'value' is false.

Macro TTI_HAS_MEMBER_FUNCTION

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/TTIntrospection.hpp>

TTI_HAS_MEMBER_FUNCTION(name)
```

Description

name = the name of the inner member.

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

The metafunction types and return:

T = the enclosing type in which to look for our 'name'.
R = the return type of the member function.
FS = an optional parameter which are the parameters of the member function as a boost::mpl forward sequence.
TAG = an optional parameter which is a boost::function_types tag to apply to the member function.
returns = 'value' is true if the 'name' exists, with the appropriate type, otherwise 'value' is false.

Macro TTI_TRAIT_HAS_MEMBER_DATA

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/TTIntrospection.hpp>

TTI_TRAIT_HAS_MEMBER_DATA(trait, name)
```

Description

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

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

The metafunction types and return:

T = the enclosing type in which to look for our 'name'.
R = the type of the member data.
returns = 'value' is true if the 'name' exists, with the appropriate type, otherwise 'value' is false.

Macro TTI_HAS_MEMBER_DATA

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/TTIntrospection.hpp>

TTI_HAS_MEMBER_DATA(name)
```

Description

name = the name of the inner member.

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

The metafunction types and return:

T = the enclosing type in which to look for our 'name'.
R = the type of the member data.
returns = 'value' is true if the 'name' exists, with the appropriate type, otherwise 'value' is false.

Macro TTI_TRAIT_HAS_STATIC_MEMBER

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

Synopsis

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

TTI_TRAIT_HAS_STATIC_MEMBER(trait, name)
```

Description

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

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

The metafunction types and return:

T = the enclosing type.
Type = the static member data or static member function type, in the form of a data or function type, in which to look for our 'name'.
returns = 'value' is true if the 'name' exists within the enclosing type, with the appropriate type, otherwise 'value' is false.

Macro TTI_HAS_STATIC_MEMBER

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

Synopsis

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

TTI_HAS_STATIC_MEMBER(name)
```

Description

name = the name of the inner member.

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

The metafunction types and return:

T = the enclosing type.
Type = the static member data or static member function type, in the form of a data or function type, in which to look for our 'name'.
returns = 'value' is true if the 'name' exists within the enclosing type, with the appropriate type, otherwise 'value' is false.

Macro TTI_TRAIT_HAS_STATIC_MEMBER_FUNCTION

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/TTIntrospection.hpp>

TTI_TRAIT_HAS_STATIC_MEMBER_FUNCTION(trait, name)
```

Description

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

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

The metafunction types and return:

T = the enclosing type in which to look for our 'name'.
R = the return type of the static member function.
FS = an optional parameter which are the parameters of the static member function as a boost::mpl forward sequence.
TAG = an optional parameter which is a boost::function_types tag to apply to the static member function.
returns = 'value' is true if the 'name' exists, with the appropriate type, otherwise 'value' is false.

Macro TTI_HAS_STATIC_MEMBER_FUNCTION

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/TTIntrospection.hpp>

TTI_HAS_STATIC_MEMBER_FUNCTION(name)
```

Description

name = the name of the inner member.

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

The metafunction types and return:

T = the enclosing type in which to look for our 'name'.
R = the return type of the static member function.
FS = an optional parameter which are the parameters of the static member function as a boost::mpl forward sequence.
TAG = an optional parameter which is a boost::function_types tag to apply to the static member function.
returns = 'value' is true if the 'name' exists, with the appropriate type, otherwise 'value' is false.

Header <boost/tti/TTIntrospectionTemplate.hpp>

```
namespace tti {
  template<typename HasTemplateCheckParams, typename T>
    struct mf_has_template_check_params;
}
```

Struct template mf_has_template_check_params

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

Synopsis

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

template<typename HasTemplateCheckParams, typename T>
struct mf_has_template_check_params {
  // types
  typedef boost::mpl::apply< HasTemplateCheckParams, typename T::type >::type type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = type::value);
};
```

Description

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

The metafunction types and return:

HasTemplateCheckParams = a Boost MPL lambda expression using the metafunction generated from either the TTI_HAS_TEMPLATE_CHECK_PARAMS (TTI_TRAIT_HAS_TEMPLATE_CHECK_PARAMS) or TTI_VM_HAS_TEMPLATE_CHECK_PARAMS (TTI_VM_TRAIT_HAS_TEMPLATE_CHECK_PARAMS) macros.
The easiest way to generate the lambda expression is to use a Boost MPL placeholder expression of the form 'metafunction<_>'.
T = The enclosing type as a nullary metafunction.

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

`mf_has_template_check_params` public member functions

1. ```
   BOOST_STATIC_CONSTANT(bool, value = type::value);
   ```

Header <boost/tti/TTIntrospectionVM.hpp>

```


TTI_VM_TRAIT_HAS_TEMPLATE_CHECK_PARAMS(trait, name, ...)
TTI_VM_HAS_TEMPLATE_CHECK_PARAMS(name, ...)
```
```

Macro TTI_VM_TRAIT_HAS_TEMPLATE_CHECK_PARAMS

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

Synopsis

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

TTI_VM_TRAIT_HAS_TEMPLATE_CHECK_PARAMS(trait, name, ...)
```

Description

trait = the name of the metafunction within the tti namespace.
name = the name of the inner class template.
... = variadic macro data which has the class template parameters.

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

The metafunction types and return:

T = the enclosing type in which to look for our 'name'.
returns = 'value' is true if the 'name' class template, with the signature as defined by the '...' variadic macro data, exists within the enclosing type, otherwise 'value' is false.

Macro TTI_VM_HAS_TEMPLATE_CHECK_PARAMS

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

Synopsis

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

TTI_VM_HAS_TEMPLATE_CHECK_PARAMS(name, ...)
```

Description

name = the name of the inner class template.
... = variadic macro data which has the class template parameters.

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

The metafunction types and return:

T = the enclosing type in which to look for our 'name'.
returns = 'value' is true if the 'name' class template, with the signature as defined by the '...' variadic macro data, exists within the enclosing type, otherwise 'value' is false.

Testing TTI

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

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

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

History

Version 1.1

- 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'.
- TTI_HAS_TYPE and 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:
  - TTI_HAS_MEMBER_DATA
  - TTI_HAS_MEMBER_FUNCTION
  - TTI_HAS_STATIC_MEMBER_FUNCTION

- New nullary type metafunction tti::mf_has_static_member_function uses the new underlying 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 tti::valid_member_type and tti::mf_valid_member_type for checking if the 'type' returned from invoking the TTI_MEMBER_TYPE or tti::mf_member_type metafunctions is valid.
- Breaking changes
  - TTI_HAS_TYPE_CHECK_TYPEDEF and tti::mf_has_type_check_typedef have been removed, and the functionality in them folded into TTI_HAS_TYPE and tti::mf_has_type.
  - TTI_MEMBER_TYPE and tti::mf_member_type no longer also return a 'valid' boolean constant. Use tti::valid_member_type or tti::mf_valid_member_type metafunctions instead (see above).
  - tti::mf_has_static_function has been removed and its functionality moved to tti::mf_has_static_member_function (see above).
  - tti::mf_member_data uses the new underlying TTI_HAS_MEMBER_DATA macro metafunction.
  - The signature for 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

- See if function templates can be introspected.
- 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 needed 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 orthogonality, 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