Boost.Utility/IdentityType 1.0.0

Lorenzo Caminiti <lorcaminiti@gmail.com>

Table of Contents

This library allows to wrap types within round parenthesis so they can always be passed as macro parameters.

Motivation

Consider the following macro which declares a variable named `varn` with the specified *`type`* (see also `var_error.cpp`):

```
#define VAR(type, n) type var ## n

VAR(int, 1);                    // OK.
VAR(std::map<int, char>, 2);    // Error.
```

The first macro invocation works correctly declaring a variable named `var1` of type `int`. However, the second macro invocation fails generating a preprocessor error similar to the following:

```
error: macro "VAR" passed 3 arguments, but takes just 2
```

That is because the `std::map` type passed as the first macro parameter contains a comma `,` not wrapped by round parenthesis `()`. The preprocessor interprets that unwrapped comma as a separation between macro parameters concluding that a total of three (and not two) parameters are passed to the macro in the following order:

1. `std::map<int`
2. `char>`
3. `2`

Note that, differently from the compiler, the preprocessor only recognizes round parenthesis `()`. Angular `<>` and squared `[]` parenthesis are not recognized by the preprocessor when parsing macro parameters.

Solution

In some cases, it might be possible to workaround this issue by avoiding to pass the type expression to the macro all together. For example, in the case above a `typedef` could have been used to specify the type expression with the commas outside the macro (see also `var.cpp`):

```
typedef std::map<int, char> map_type;
VAR(map_type, 3); // OK.
```

When this is neither possible nor desired (e.g., see the function template `f` in the section below), this library header `boost/utility/identity_type.hpp` defines a macro `BOOST_IDENTITY_TYPE` which can be used to workaround the issue while keeping the type expression as one of the macro parameters (see also `var.cpp`).

```
#include <boost/utility/identity_type.hpp>

VAR(BOOST_IDENTITY_TYPE((std::map<int, char>)), 4); // OK.
```

The `BOOST_IDENTITY_TYPE` macro expands to an expression that evaluates (at compile-time) to the specified type. The specified type is never split into multiple macro parameters because it is always wrapped by a set of extra round parenthesis `()`. In fact, a total of two sets of round parenthesis must be used: The parenthesis to invoke the macro `BOOST_IDENTITY_TYPE(...)` plus the inner parenthesis to wrap the type passed to the macro `BOOST_IDENTITY_TYPE((...))`.

This macro works on any C++03 compiler (and it does not use variadic macros). [1] The authors originally developed and tested this library using GNU Compiler Collection (GCC) C++ 4.5.3 (with and without C++11 features enabled `-std=c++0x`) on Cygwin and Miscrosoft Visual C++ (MSVC) 8.0 on Windows 7. See the library regressions test results for more information on supported compilers and platforms.

[1] Using variadic macros, it would be possible to require a single set of extra parenthesis `BOOST_IDENTITY_TYPE(`*`type`*`)` instead of two `BOOST_IDENTITY_TYPE((`*`type`*`))` but variadic macros are not part of C++03 (even if nowadays they are supported by most modern compilers and they are also part of C++11).

Templates

This macro must be prefixed by `typename` when used within templates. For example, let's program a macro that declares a function parameter named `arg`*n* with the specified *type* (see also `template.cpp`):

```
#define ARG(type, n) type arg ## n

template<typename T>
void f( // Prefix macro with `typename` in templates.
    ARG(typename BOOST_IDENTITY_TYPE((std::map<int, T>)), 1)
) {
    std::cout << arg1[0] << std::endl;
}
```

```
std::map<int, char> a;
a[0] = 'a';

f<char>(a); // OK...
// f(a);    // ... but error.
```

However, note that the template parameter `char` must be manually specified when invoking the function as in `f<char>(a)`. In fact, when the `BOOST_IDENTITY_TYPE` macro is used to wrap a function template parameter, the template parameter can no longer be automatically deduced by the compiler form the function call as `f(a)` would have done. [2] (This limitation does not apply to class templates because class template parameters must always be explicitly specified.) In other words, without using the `BOOST_IDENTITY_TYPE` macro, C++ would normally be able to automatically deduce the function template parameter as shown below:

```
template<typename T>
void g(
    std::map<int, T> arg1
) {
    std::cout << arg1[0] << std::endl;
}
```

```
g<char>(a); // OK...
g(a);       // ... and also OK.
```

[2] This is because the implementation of `BOOST_IDENTITY_TYPE` wraps the specified type within a meta-function.

Abstract Types

On some compilers (e.g., GCC), using this macro on abstract types (i.e., classes with one or more pure virtual functions) generates a compiler error. This can be avoided by manipulating the type adding and removing a reference to it.

Let's program a macro that performs a static assertion on a Template Meta-Programming (TMP) meta-function (similarly to Boost.MPL `BOOST_MPL_ASSERT`). The `BOOST_IDENTITY_TYPE` macro can be used to pass a meta-function with multiple template parameters to the assert macro (so to handle the commas separating the template parameters). In this case, if the meta-function is an abstract type, it needs to be manipulated adding and removing a reference to it (see also `abstract.cpp`):

```
#define TMP_ASSERT(metafunction) \
    BOOST_STATIC_ASSERT(metafunction::value)

template<typename T, bool b>
struct abstract {
    static const bool value = b;
    virtual void f(T const& x) = 0;     // Pure virtual function.
};

TMP_ASSERT(
    boost::remove_reference<            // Add and remove
        BOOST_IDENTITY_TYPE((           // reference for
            boost::add_reference<       // abstract type.
                abstract<int, true>
            >::type
        ))
    >::type
);
```

Annex: Usage

The BOOST_IDENTITY_TYPE macro can be used either when calling a user-defined macro (as shown by the examples so far), or internally when implementing a user-defined macro (as shown below). When BOOST_IDENTITY_TYPE is used in the implementation of the user-defined macro, the caller of the user macro will have to specify the extra parenthesis (see also paren.cpp):

```
#define TMP_ASSERT_PAREN(parenthesized_metafunction) \
    /* use `BOOST_IDENTITY_TYPE` in macro definition instead of invocation */ \
    BOOST_STATIC_ASSERT(BOOST_IDENTITY_TYPE(parenthesized_metafunction)::value)

#define TMP_ASSERT(metafunction) \
    BOOST_STATIC_ASSERT(metafunction::value)

// Specify only extra parenthesis `((...))`.
TMP_ASSERT_PAREN((boost::is_const<std::map<int, char> const>));

// Specify both the extra parenthesis `((...))` and `BOOST_IDENTITY_TYPE` macro.
TMP_ASSERT(BOOST_IDENTITY_TYPE((boost::is_const<std::map<int, char> const>)));
```

However, note that the caller will *always* have to specify the extra parenthesis even when the macro parameters contain no comma:

```
TMP_ASSERT_PAREN((boost::is_const<int const>)); // Always extra `((...))`.

TMP_ASSERT(boost::is_const<int const>); // No extra `((...))` and no macro.
```

In some cases, using BOOST_IDENTITY_TYPE in the implementation of the user-defined macro might provide the best syntax for the caller. For example, this is the case for BOOST_MPL_ASSERT because the majority of template meta-programming expressions contain unwrapped commas so it is less confusing for the user to always specify the extra parenthesis ((...)) instead of using BOOST_IDENTITY_TYPE:

```
BOOST_MPL_ASSERT(( // Natural syntax.
    boost::mpl::and_<
          boost::is_const<T>
        , boost::is_reference<T>
    >
));
```

However, in other situations it might be preferable to not require the extra parenthesis in the common cases and handle commas as special cases using BOOST_IDENTITY_TYPE. For example, this is the case for BOOST_LOCAL_FUNCTION for which always requiring the extra parenthesis ((...)) around the types would lead to an unnatural syntax for the local function signature:

```
int BOOST_LOCAL_FUNCTION( ((int&)) x, ((int&)) y ) { // Unnatural syntax.
    return x + y;
} BOOST_LOCAL_FUNCTION_NAME(add)
```

Instead requiring the user to specify BOOST_IDENTITY_TYPE only when needed allows for the more natural syntax BOOST_LOCAL_FUNCTION(int& x, int& y) in the common cases when the parameter types contain no comma (while still allowing to specify parameter types with commas as special cases using BOOST_LOCAL_FUNCTION(BOOST_IDENTITY_TYPE((std::map<int, char>))& x, int& y)).

Annex: Implementation

The implementation of this library macro is equivalent to the following: [3]

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

#define BOOST_IDENTITY_TYPE(parenthesized_type) \
    boost::function_traits<void parenthesized_type>::arg1_type
```

Essentially, the type is wrapped between round parenthesis `(std::map<int, char>)` so it can be passed as a single macro parameter even if it contains commas. Then the parenthesized type is transformed into the type of a function returning `void` and with the specified type as the type of the first and only argument `void (std::map<int, char>)`. Finally, the type of the first argument `arg1_type` is extracted at compile-time using the `function_traits` meta-function therefore obtaining the original type from the parenthesized type (effectively stripping the extra parenthesis from around the specified type).

[3] There is absolutely no guarantee that the macro is actually implemented using the code listed in this documentation. The listed code is for explanatory purposes only.

Reference

Header <boost/utility/identity_type.hpp>

Wrap type expressions with round parenthesis so they can be passed to macros even if they contain commas.

```
BOOST_IDENTITY_TYPE(parenthesized_type)
```

Macro BOOST_IDENTITY_TYPE

BOOST_IDENTITY_TYPE — This macro allows to wrap the specified type expression within extra round parenthesis so the type can be passed as a single macro parameter even if it contains commas (not already wrapped within round parenthesis).

Synopsis

```
// In header: <boost/utility/identity_type.hpp>

BOOST_IDENTITY_TYPE(parenthesized_type)
```

Description

Parameters:

`parenthesized_type`	The type expression to be passed as macro parameter wrapped by a single set of round parenthesis `( . . . )`. This type expression can contain an arbitrary number of commas.

This macro works on any C++03 compiler (it does not use variadic macros).

This macro must be prefixed by `typename` when used within templates. Note that the compiler will not be able to automatically determine function template parameters when they are wrapped with this macro (these parameters need to be explicitly specified when calling the function template).

On some compilers (like GCC), using this macro on abstract types requires to add and remove a reference to the specified type.