The Variadic Macro Data Library 1.7

Edward Diener

Table of Contents

Introduction

Welcome to version 1.7 of the Variadic Macro Data library.

The Variadic Macro Data library, referred to hereafter as VMD for short, is a library of variadic macros which provide enhancements to the functionality in the Boost preprocessor library (Boost PP), especially as it relates to preprocessor "data types".

The functionality of the library may be summed up as:

1. Provide a better way of testing for and using empty parameters and empty preprocessor data.

2. Provide ways for testing/parsing for identifiers, numbers, tuples, arrays, lists. and seqs.

3. Provide ways for testing/parsing sequences of identifiers, numbers, tuples, arrays, lists. and seqs.

4. Provide some useful variadic macros not in Boost PP.

The library is a header only library and all macros in the library are included by a single header, whose name is 'vmd.hpp'. Individual headers may be used for different functionality in the library and will be denoted when that functionality is explained.

All the macros in the library begin with the sequence 'BOOST_VMD_' to distinguish them from other macros the end-user might use. Therefore the end-user should not use any C++ identifiers, whether in macros or otherwise, which being with the sequence 'BOOST_VMD_'.

Use of the library is only dependent on Boost PP. The library also uses Boost detail lightweight_test.hpp for its own tests. The library currently depends on the 'develop' branch of the Boost PP library in the modular boost structure. Once Boost 1.56 is released the 'develop' branch of Boost PP in modular-boost will be merged to the 'master' branch.

The library is NOT a Boost library but it is in the Boost review queue and is eagerly looking for a review manager. Please contact Edward Diener at eldiener@tropicsoft.com if you are interested as serving as the review manager for the library.

Variadic Macros

Variadic macros, as specified by C++11, is a feature taken from the C99 specification. They are macros which take a final parameter denoted as '...' which represents one or more final arguments to the macro as a series of comma-separated tokens. In the macro expansion a special keyword of '__VA_ARGS__' represents the comma-separated tokens. This information when passed to a variadic macro I call 'variadic macro data', which gives its name to this library. The more general term 'variadic data' is used in this documentation to specify data passed to a macro which can contain any number of macro tokens as a single macro parameter, such as is found in Boost PP data types.

Boost support

The Boost PP library has support for variadic macros and uses its own criteria to determine if a particular compiler has that support. Boost PP uses the macro BOOST_PP_VARIADICS to denote whether the compiler being used supports variadic macros. When BOOST_PP_VARIADICS is set to 1 the compiler supports variadic macros, otherwise when BOOST_PP_VARIADICS is set to 0 the compiler does not support variadic macros. If a user of Boost PP sets this value, Boost PP uses the value the end-user sets. This macro can also be checked to determine if a compiler has support for variadic macros.

Determining variadic macro support

The VMD library automatically determines whether variadic macro support is enabled for a particular compiler by also using the same BOOST_PP_VARIADICS macro from Boost PP. The end-user of VMD can also manually set the macro BOOST_PP_VARIADICS to turn on or off compiler support for variadic macros in the VMD library. When BOOST_PP_VARIADICS is set to 0 variadic macros are not supported in the VMD library, otherwise when BOOST_PP_VARIADICS is set to non-zero they are supported in the VMD library. This same macro can be used to determine if VMD supports variadic macros for a particular compiler.

Since this library depends on variadic macro support, if BOOST_PP_VARIADICS is set to 0, using any of the macros in VMD will lead to a compiler error since the macro will not be defined. However just including any of the header files in VMD, even with no variadic macro support for the compiler, will not lead to any compiler errors.

Naming conventions

All of the macros in the library begin with the prefix BOOST_VMD_, where VMD stands for 'Variadic Macro Data'.

Following the prefix, certain names in the macros refer to data types in this library or Boost PP. These names and their data types are:

1. TUPLE = Boost PP tuple data type.
2. ARRAY = Boost PP array data type.
3. LIST = Boost PP list data type.
4. SEQ = Boost PP seq data type.
5. IDENTIFIER = A VMD identifier
6. NUMBER = A VMD number

I have used most of these names in order to mimic the naming of Boost PP as closely as possible. Subsequent use of the words 'tuple', 'array', 'list', and 'seq' refer to these Boost PP data types unless otherwise noted. See the help for Boost PP for any explanation of these data types. For a VMD identifier I use the term 'v-identifier' and for a VMD number I use the term 'v-number'. Documentation for these in VMD will follow.

Functional groups

The particular constructs for which VMD has functionality can be divided into these categories:

1. Emptiness
2. Identifiers
3. Numbers
4. Boost PP data types (array, list, seq, and tuple)
5. Data sequences
6. Additional variadic macros

A further general explanation of each of these categories follow in the appropriate place.

Visual C++

Microsoft's Visual C++ compiler, abbreviated VC++, is a very popular compiler but does not implement the standard C++ preprocessor correctly in a number of respects. Because of this the programmer using the VMD needs to occasionally do things differently when VC++ is being used. These "quirks" of VC++ have been smoothed over as much as possible in the VMD library, but are mentioned in further topics and occasionally must be addressed by the programmer using VMD.

The VMD has a macro that indicates when VC++ is the compiler being used. The macro is an object-like macro called BOOST_VMD_MSVC. It is set to 1 when VC++ is being used and set to 0 when VC++ is not being used.

Emptiness

Passing empty arguments

It is possible to pass an empty argument to a macro. The official terminology for this in the C++ standard is an argument "consisting of no preprocessing tokens".

Let us consider a number of cases without worrying too much what the macro output represents.

Consider these two function-like macros:

```
#define SMACRO() someoutput
#define EMACRO(x) otheroutput x
```

The first macro takes no parameters so invoking it must always be done by

```
SMACRO()
```

and passing any arguments to it would be invalid.

The second macro takes a single parameter. it can be evoked as

```
EMACRO(somedata)
```

but it also can be invoked as

```
EMACRO()
```

In the second invocation of EMACRO we are passing an empty argument to the macro. Similarly for any macro having 1 or more parameters, an empty argument can be validly passed for any of the parameters, as in

```
#define MMACRO(x,y,z) x y z

MMACRO(1,,2)
```

An empty argument is an argument even if we are passing nothing.

Now just because an empty argument can be passed for a given parameter of a macro does not mean one should do so. Any given macro will specify what each argument to a macro should represent, and it is normally very rare to encounter a macro which specifies that an empty argument can logically be passed for a given argument. But from the perspective of standard C++ it is perfectly valid to pass an empty argument for a macro parameter.

The notion of passing empty arguments can be extended to passing empty data which "consists of no preprocessing tokens" in slightly more complicated situations. It is possible to pass empty data as an argument to a variadic macro in the form of variadic macro data, as in

```
#define VMACRO(x,...) x __VA_ARGS__
```

invoked as

```
VMACRO(somedata,)
```

Here one passes empty data as the variadic maco data and it is perfectly valid C++. Please notice that this different from

```
VMACRO(somedata)
```

which is not valid C++ since something must be passed for the variadic argument. Similar one could invoke the macro as

```
VMACRO(somedata,vdata1,,vdata3)
```

where one is passing variadic macro data but an element in the variadic macro data is empty.

Furthermore if we are invoking a macro which expects a Boost PP data type, such as a tuple, we could also validly pass empty data for all or part of the data in a tuple, as in

```
#define TMACRO(x,atuple) x atuple

TMACRO(somedata,())
```

In this case we are passing a 1 element tuple where the single element itself is empty.

or

```
TMACRO(somedata,(telem1,,telem2,teleem3))
```

In this case we are passing a 4 element tuple where the second element is empty.

Again either invocation is valid C++ but it is usually not what the designed of the macro has desired, even if in both cases the macro designer has specified that the second parameter must be a tuple for the macro to work properly.

Returning emptiness

Similar to passing empty arguments in various ways to a macro, the data which a macro returns (or 'generates' may be a better term) could be empty, in various ways. Again I am not necessarily promoting this idea a a common occurrence of macro design but merely pointing it out as valid C++ preprocessing.

```
#define RMACRO(x,y,z)

RMACRO(data1,data2,data3)
```

It is perfectly valid C++ to return "nothing" from a macro invocation. In fact a number of macros in Boost PP do that based on the preprocessor metaprogramming logic of the macro, and are documented as such.

Similarly one could return nothing as part or all of a Boost PP data type or even as part of variadic macro data.

```
#define TRETMACRO(x,y,z) ()
#define TRETMACRO1(x,y,z) (x,,y,,z)
#define VRETMACRO(x,y,z) x,,y,,z
```

Here again we are returning something but in terms of a Boost PP tuple or in terms of variadic data, we have elements which are empty.

Emptiness in preprocessor metaprogramming

In the examples given above where "emptiness" in one form of another is passed as arguments to a macro or returned from a macro, the examples I have given were created as simplified as possible to illustrate my points. In actual preprocessor metaprogramming, using Boost PP, where complicated logic is used to generate macro output based on the arguments to a macro, it might be useful to allow and work with empty data if one were able to test for the fact that data was indeed empty.

Testing for empty data

Currently Boost PP has an undocumented macro for testing whether a parameter is empty of not, written without the use of variadic macros. The macro is called BOOST_PP_IS_EMPTY. The macro is by its nature flawed, since there is no generalized way of determining whether or not a parameter is empty using the C++ preprocessor. But the macro will work given input limited in various ways or having actual emptiness. Probably because the macro is flawed, and because there really is no perfect way of testing for emptiness in the C++ preprocessor, the Boost PP library did not want to document this macro for use by others.

Paul Mensonides, the developer of Boost PP and the BOOST_PP_IS_EMPTY macro in that library, also wrote a better macro using variadic macros, for determining whether or not a parameter is empty or not, which he published on the Internet in response to a discussion about emptiness. This macro is also not perfect, since there is no perfect solution, but will work correctly with almost all input. I have adapted his code for the VMD and developed my own very slightly different code.

The macro is called `BOOST_VMD_IS_EMPTY` and will return 1 if its input is empty or 0 if its input is not empty. The macro is a variadic macro which make take any input [1].

Macro Flaw with a standard C++ compiler

The one situation where the macro does not work properly is if its input resolves to a function-like macro name or a sequence of preprocessor tokens ending with a function-like macro name and the function-like macro takes two or more parameters.

Here is a simple example:

```
#define FMACRO(x,y) any_output

BOOST_VMD_IS_EMPTY(FMACRO)
BOOST_VMD_IS_EMPTY(some_input FMACRO)
```

In the first case the name of a function-like macro is being passed to BOOST_VMD_IS_EMPTY while in the second case a sequence of preprocessing tokens is being passed to BOOST_VMD_IS_EMPTY ending with the name of a function-like macro. The function-like macro also has two (or more) parameters. In both the cases above a compiler error will result from the use of BOOST_VMD_IS_EMPTY.

Please note that these two problematical cases are not the same as passing an invocation of a function-like macro name to BOOST_VMD_IS_EMPTY, as in

```
BOOST_VMD_IS_EMPTY(FMACRO(arg1,arg2))
BOOST_VMD_IS_EMPTY(someinput FMACRO(arg1,arg2))
```

which always works correctly, unless of course a particular function-like macro invocation resolves to either of our two previous situations.

So for a standard comforming compiler we have essentially a single corner case where the BOOST_VMD_IS_EMPTY does not work and, when it does not work it, produces a compiler error rather than an incorrect result.

Macro Flaw with Visual C++

The VC++ preprocessor is not a standard C++ conforming preprocessor in at least two relevant situations. These situations combine to create a single corner case which causes the BOOST_VMD_IS_EMPTY macro to not work properly using VC++ when the input resolves to a function-like macro name.

The first situation is that if a macro taking 'n' number of parameters is invoked with 0 to 'n-1' parameters, the compiler does not give an error, but only a warning.

[1] For VC++ 8 the input is not variadic data but a single parameter

```
#define FMACRO(x,y) x + y

FMACRO(1)
```

should give a compiler error, as it does when using a C++ standard-conforming compiler, but when invoked using VC++ it only gives a warning and VC++ continues macro substitution with 'y' as a placemarker preprocessing token. This non-standard conforming action actually eliminates the case where BOOST_VMD_IS_EMPTY does not work properly with a standard C++ conforming compiler. But of course it has the potential of producing incorrect output in other macro processing situations unrelated to the BOOST_VMD_IS_EMPTY invocation, where a compiler error should occur.

A second general situation with VC++ situation, which affects the use of BOOST_VMD_IS_EMPTY, is that the expansion of a macro works incorrectly when the expanded macro is a function-like macro name followed by a function-like macro invocation, in which case the macro re-expansion is erroneously done more than once. This latter case can be seen by this example:

```
#define FMACRO1(parameter) FMACRO3 parameter()
#define FMACRO2() ()
#define FMACRO3() 1

FMACRO1(FMACRO2)

should expand to:

FMACRO3()

but in VC++ it expands to:

1
```

where after initially expanding the macro to:

```
FMACRO3 FMACRO2()
```

VC++ erroneously rescans the sequence of preprocessing tokens more than once rather than rescan just one more time for more macro names.

What these two particular preprocessor flaws in the VC++ compiler mean is that although BOOST_VMD_IS_EMPTY does not fail with a compiler error in the same case as with a standard C++ conforming compiler given previously, it fails by giving the wrong result in another situation.

The failing situation is:

when the input to BOOST_VMD_IS_EMPTY resolves to only a function-like macro name taking any number of parameters, and the function-like macro, when passed a single empty argument, expands to a tuple, BOOST_VMD_IS_EMPTY will erroneously return 1 rather than 0 when using the Visual C++ compiler.

Here is an example of the failure:

```
#define FMACRO(any_number_of_parameters) ( any_number_of_tuple_elements )

BOOST_VMD_IS_EMPTY(FMACRO) // erroneously returns 1, instead of 0
```

As with a standard C++ conforming compiler, we have a rare corner case where the BOOST_VMD_IS_EMPTY will not work properly, but unfortunately in this very similar but even rarer corner case with VC++, we will silently get an incorrect result rather than a compiler error.

I want to reiterate that there is no perfect solution in C++ to the detection of emptiness even for a C++ compiler whose preprocessor is complete conformant, which VC++ obviously is not.

Macro Flaw conclusion

With all of the above mentioned, the case(s) where BOOST_VMD_IS_EMPTY will work incorrectly are very small, even with the erroneous VC++ preprocessor, and I consider the macro worthwhile to use since it works correctly with the vast majority of possible preprocessor input.

The case where it will not work, with both a C++ standard confoming preprocessor or with Visual C++, occurs when he name of a function-like macro is part of the input to BOOST_VMD_IS_EMPTY. Obviously the macro should be used by the preprocessor metaprogrammer when the possible input to it is constrained to eliminate the erroneous case.

Furthermore, since emptiness can correctly be tested for in nearly every situation, the macro can be used internally when the preprocessor metaprogrammer wants to return data from a macro and all or part of that data could be empty.

Therefore I believe the macro is quite useful, despite the corner case flaw which makes it imperfect. Consequently I believe that the preprocessor metaprogrammer can use the concept of empty preprocessor data in the design of his own macros.

Using the macro

The macro BOOST_VMD_IS_EMPTY is used extensively throughout VMD and macro programmers may find this macro useful in their own programming efforts despite the slight flaw in the way that it works. Explanations of when the macro is used by other macros throughout the VMD library, and how to avoid the flaw in BOOST_VMD_IS_EMPTY, will be given for each macro in the library which uses BOOST_VMD_IS_EMPTY.

The end-user of VMD can include the individual header file 'is_empty.hpp' instead of the general header file 'vmd.hpp' for using this macro.

Macro constraints

When discussing the BOOST_VMD_IS_EMPTY macro I mentioned constraining input to the macro. Now I will discuss what this means in terms of preprocessor metaprogramming and input to macros in general.

Constrained input

When a programmer designs any kinds of callables in C++ (functions, member functions etc.), he specifies what the types of input and the return value are. The C++ compiler enforces this specification at compile time. Similarly at run-time a callable may check that its input falls within certain documented and defined boundaries and react accordingly if it does not. This is all part of the constraints for any callable in C++ and should be documented by any good programmer.

The C++ preprocessor is much "dumber" than the C++ compiler and even with the preprocessor metaprogramming constructs which Paul Mensonides has created in Boost PP there is far less the preprocessor metaprogrammer can do at preprocessing time to constrain argument input to a macro than a programmer can do at compile-time and/or at run-time to constrain argument input to a C++ callable. Nevertheless it is perfectly valid to document what a macro expects as its argument input and, if a programmer does not follow the constraint, the macro will fail to work properly. In the ideal case in preprocessor metaprogramming the macro could tell whether or not the constraint was met and could issue some sort of intelligible preprocessor error when this occured, but even within the reality of preprocessor metaprogramming with Boost PP this is not always possible to do. Nevertheless if the user of a macro does not follow the constraints for a macro parameter, as specified in the documentation of a particular macro being invoked, any error which occurs is the fault of that user. I realize that this may go against the strongly held concept that programming errors must always be met with some sort of compile-time or run-time occurrence which allows the programmer to correct the error, rather than a silent failure which masks the error. Because the preprocessor is "dumber" and cannot provide this occurrence in all cases the error could unfortunately be masked, despite the fact that the documentation specifies the correct input constraint(s). In the case of the already discussed macro BOOST_VMD_IS_EMPTY, this masking of the error only occurs with a preprocessor (Visual C++) which is not C++ standard conformant.

The Boost PP library does have a way of generating a preprocessor error, without generating preprocessor output, but once again this way does not work with the non-conformant preprocessor of Visual C++. The means to do so using Boost PP is through the BOOST_PP_ASSERT macro. As will be seen and discussed later VMD has an equivalent macro which will work with Visual C++ by producing incorrect C++ output rather than a preprocessing error, but even this is not a complete solution since the incorrect C++ output produced could be hidden.

Even the effort to produce a preprocessor error, or incorrect output inducing a compile-time error, does not solve the problem of constrained input for preprocessor metaprogramming. Often it is impossible to determine if the input meets the constraints which the preprocessor metaprogrammer places on it and documents. Certain preprocessing tokens cannot be checked reliably for particular values, or a range of values, without the checking mechanism itself possibly creating a preprocessing error.

This does not mean that one should give up attempting to check macro input constraints. If it can be done I see the value of such checks and a number of VMD macros, discussed later, are designed as preprocessing input constraint checking macros. But the most important thing when dealing with macro input constraints is that they should be carefully documented and that the programmer should know that if the constraints are not met, either preprocessing errors or incorrect macro results could be the results.

In this the VMD library is less conservative than the Boost PP library. Paul Mensonides design for Boost PP almost always means that the greatest care has been taken to not produce erroneous results, even in the face of erroneous input. The VMD library, in order to present more preprocessor programming functionality and flexibility, allows that erroneous results could occur if certain input constraints are not met, whether the erroneous results are preprocessor errors or incorrect output from a VMD macro. At the same time the VMD does everything that the preprocessor is capable of doing to check the input constraints, and carefully documents for each macro in the library what the input for each could be in order to avoid erroneous output.

I believe that documented macro input constraints are just as valid in the preprocessor as compile-time/run-time constraints are valid in C++, even if the detection of such constraints and/or the handling of constraints that are not met are far more difficult, if not impossible, in the preprocessor than in the compile-time/run-time processing of C++.

The VMD library uses constraints for most of it macros and the documentation for those macros mentions the constraints that apply in order to use the macro.

Data types

The VMD library has functionality for testing and parsing preprocessor "data types".

The C++ preprocessor defines preprocessor data as preprocessing tokens. The types of preprocessing tokens can be seen in section 2.5 of the C++ standard document.

The VMD library works with a subset of two of these types of preprocessor tokens as "data types". These are the "identifier" and "pp-number" preprocessor tokens.

VMD identifiers, or 'v-identifiers' for short, are sequences of preprocessing tokens consisting of alphanumeric characters and the underscore (_) character. This is very similar to a preprocessor token identifier with the difference being that a v-identifier can start with a numeric character, allowing v-identifiers to also be positive integral literals. VMD offers functionality for parsing v-identifiers both as a separate element or in a sequence of preprocessing tokens.

VMD numbers, or 'v-numbers' for short, are Boost PP numbers, ie. preprocessing tokens of whole numbers between 0 and 256 inclusive. These are a small subset of preprocessor token "pp-numbers". VMD offers functionality for parsing v-numbers both as a separate element or in a sequence of preprocessing tokens. A v-number is really a closed set of v-identifiers for which VMD offers specific functionality.

The Boost PP library supports four individual high-level data types. These are arrays, lists, seqs, and tuples, although when using variadic macros arrays are really obsolete since tuples have all the functionality of arrays with a simpler syntax. A further data type supported by Boost PP is variadic data, which is a comma separated grouping of preprocessor elements.

VMD has functionality to work with the four Boost PP high-level data types. VMD can test the Boost PP data types and parse them in a sequence of preprocessor tokens.

The ability to identify emptiness as well as these six types of "data types" represents a large part of the functionality which VMD has to offer. I will collectively be calling these "data types" by the name of "v-types" in the subsequent documentation.

Identifiers

An identifier in VMD, hereafter called a 'v-identifier', is either of two possibilities:

- a preprocessing token 'identifier', which is essentially a sequence of alphanumeric characters and the underscore character with the first character not being a numeric character.
- a preprocessing token 'pp-number' that is an integral literal token.

Here are some examples:

```
SOME_NAME
_SOME_NAME
SOME_123_NAME
some_123_name
sOMe_123_NAmE
2367
43e11
0
22
654792
0x1256
```

One of the difficulties with identifiers in preprocessor metaprogramming is safely testing for a particular one. VMD has a means of doing this within a particular constraint for the characters that serve as the input.

The constraint is that the beginning input character, ignoring any whitespace, passed as the input to test must be either:

- an identifier character, ie. an alphanumeric or an underscore
- the left parenthesis of a tuple.

If this is not the case a preprocessing error will occur.

Given the input:

```
s_anything : can be tested
S_anything : can be tested
s_anYthiNg : can be tested
_anything : can be tested
_Anything : can be tested
_anytHIng : can be tested
24 : can be tested
245e2: can be tested
(anything) : can be tested, tuple
(anything) anything : can be tested, tuple and further input

%_anything : will give a preprocessing error due to the constraint
(_anything : will give a preprocessing error due to the con↵
straint, since a single '(' does not form a tuple
```

The macro used to test for a particular identifier in VMD is called BOOST_VMD_IS_IDENTIFIER. The macro takes two parameters. The first parameter is the input to test against, while the second parameter is the 'key', hereafter called a v-key. The v-key can be used to test against any one of a number of possible identifiers, since the v-key can be either a single token or a tuple of tokens.

The form of a v-key is that of a v-identifier itself, except that the v-key should not begin with an underscore character (_).

In order to use a v-key the programmer invoking this macro must define an object-like macro, called a key-identifier macro, whose form is:

```
#define BOOST_VMD_MAP_'v-key''v-identifier'
```

A key-identifier macro must generate no output. The v-key portion must be unique, for each identifier to be tested, and which will not be duplicated within the translation unit. Two of the easiest ways to create a unique v-key across the translation unit is:

- preface all v-keys with a single unique preface for the library or application in which BOOST_VMD_IS_IDENTIFIER is being used, followed by some distinct v-key name.
- generate a GUID and use its hex representation as the unique v-key.

The programmer can define any number of key-identifier macros, one for each identifier he wishes to test. Once the key-identifier macro is defined, the v-key is passed as the second parameter to BOOST_VMD_IS_IDENTIFIER to see if the first parameter is the same as the v-identifier.

Since the second parameter can be a single token or a tuple of tokens, the BOOST_VMD_IS_IDENTIFIER macro returns a number, from 0 to the number of tokens passed. A return of 0 indicates that the v-identifier was not found, while a return of a non-zero indicates the v-identifier was found and, in the case of multiple tokens passed as a tuple, indicates which one of the v-identifiers was found (1 for the first token, 2 for the second token, etc.). The programmer can then use a Boost PP comparison macro, such as BOOST_PP_EQUAL, to determine which v-identifier is found. The programmer can also use BOOST_PP_IF or BOOST_PP_IIF (with a single token v-key) with the result to branch as part of his preprocessor metaprogramming logic.

Example

Let us look at an example of how to use BOOST_VMD_IS_IDENTIFIER.

```
#define BOOST_VMD_MAP_MYLIB_KEY1_CIRCLE
#define BOOST_VMD_MAP_MYLIB_KEY2_SQUARE
#define BOOST_VMD_MAP_MYLIB_KEY3_TRIANGLE
#define BOOST_VMD_MAP_MYLIB_KEY4_RECTANGLE

BOOST_VMD_IS_IDENTIFIER(input,(MYLIB_KEY1_,MYLIB_KEY2_,MYLIB_KEY3_,MYLIB_KEY4_))

returns:

if input = CIRCLE, 1
if input = SQUARE, 2
if input = TRIANGLE, 3
if input = RECTANGLE, 4
if input = PARALLELOGRAM, 0 since there is no match of any of the keys
if input = CIRCLE DATA, 0 since there are tokens after the identifier
if input = %NAME, does not meet the constraint therefore a preprocessing error occurs
if input = ( NAME ) NAME, 0 since the macro begins with a tuple and this can be tested for
```

Usage

To use the BOOST_VMD_IS_IDENTIFIER macro either include the general header:

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

or include the specific header:

```
#include <boost/vmd/is_identifier.hpp>
```

Numbers

A number in VMD, herafter called a 'v-number', is a preprocessing 'pp-number', limited to a Boost PP number. This is an integral literal between 0 and 256. The form of the number does not contain leading zeros. Acceptable as v-numbers are:

```
0
127
33
254
18
```

but not:

```
033
06
009
00
```

As can be seen from the explanation of a v-identifier, a v-number is merely a small subset of all possible v-identifiers, for which VMD internally provides key-identifier macros to test. Therefore the particular constraint on the input to test is exactly the same as for v-identifiers.

The constraint is that the beginning input character, ignoring any whitespace, passed as the input to test must be either:

- an identifier character, ie. an alphanumeric or an underscore
- the left parenthesis of a tuple.

If this is not the case a preprocessing error will occur.

Given the input:

```
s_anything : can be tested
S_anything : can be tested
s_anYthiNg : can be tested
s&_anYthiNg : can be tested
_anything : can be tested
_Anything : can be tested
_anytH?Ing : can be tested
24 : can be tested
245e2: can be tested
(anything) : can be tested, tuple
(anything) anything : can be tested, tuple + identifier

%_anything : will give a preprocessing error due to the constraint
(_anything : will give a preprocessing error due to the con↵
straint, since a single '(' does not form a tuple
```

The macro used to test for a particular v-number in VMD is called BOOST_VMD_IS_NUMBER. The macro takes a single parameter, the input to test against.

The macro returns 1 if the parameter is a Boost PP number, otherwise the maco returns 0.

Example

Let us look at an example of how to use BOOST_VMD_IS_NUMBER.

```
BOOST_VMD_IS_NUMBER(input)

returns:

if input = 0, 1
if input = 44, 1
if input = SQUARE, 0
if input = 44 DATA, 0 since there are tokens after the number
if input = 044, 0 since no leading zeros are allowed for our Boost PP numbers
if input = 256, 1
if input = 257, 0 since it falls outside the Boost PP number range
if input = %44, does not meet the constraint therefore a preprocessing error occurs
if input = 44.0, 0 since the number is a floating point literal
if input = ( 44 ), 0 since the macro begins with a tuple and this can be tested for
```

Example

To use the BOOST_VMD_IS_NUMBER macro either include the general header:

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

or include the specific header:

```
#include <boost/vmd/is_number.hpp>
```

Boost PP data types

VMD is able to determine whether or not preprocessing input is a given Boost PP data type. The VMD macros to do this are:

- BOOST_VMD_IS_ARRAY for an array
- BOOST_VMD_IS_LIST for a list
- BOOST_VMD_IS_SEQ for a seq
- BOOST_VMD_IS_TUPLE for a tuple

Each of these macros take a single parameter as input and return 1 if the parameter is the appropriate data type and 0 if it is not.

One thing to realize is that both an array and a non-empty list are also a tuple. So if one has:

```
#define ANARRAY (3,(a,b,c))
#define ALIST (a,(b,(c,BOOST_PP_NIL)))
#define ATUPLE (a,b,c)
#define ASEQ (a)(b)(c)
```

then

```
BOOST_VMD_IS_TUPLE(ANARRAY) returns 1
BOOST_VMD_IS_TUPLE(ALIST) returns 1
BOOST_VMD_IS_TUPLE(ATUPLE) returns 1
BOOST_VMD_IS_TUPLE(ASEQ) returns 0
```

Another thing to realize is that a single element tuple is also a one element seq. So if one has:

```
#define ASE_TUPLE (a)
```

then

```
BOOST_VMD_IS_TUPLE(ASE_TUPLE) returns 1
BOOST_VMD_IS_SEQ(ASE_TUPLE) returns 1
```

Unlike the processing of v-identifiers or v-numbers, there is no constraint about the beginning of the input causing a preprocessing error since preprocessor concatenation is not used to determine any of these data types.

Usage

You can use the geneal header file:

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

or you can use individual header files for each of these macros. The individual header files are:

```
#include <boost/vmd/array.hpp> //  for the BOOST_VMD_IS_ARRAY macro
#include <boost/vmd/list.hpp> //  for the BOOST_VMD_IS_LIST macro
#include <boost/vmd/seq.hpp> //  for the BOOST_VMD_IS_SEQ macro
#include <boost/vmd/tuple.hpp> // for the BOOST_VMD_IS_TUPLE macro.
```

Identifying data types

Identifying macros and BOOST_VMD_IS_EMPTY

The various macros for identifying VMD data types complement the ability to identify emptiness using BOOST_VMD_IS_EMPTY. They also share with BOOST_VMD_IS_EMPTY the inherent flaw mentioned when discussing this macro, since they themselves use BOOST_VMD_IS_EMPTY to determine that the input has ended.

To recapitulate the flaw with BOOST_VMD_IS_EMPTY:

- using a standard C++ compiler if the input ends with the name of a function-like macro, and that macro takes two or more parameters, a preprocessing error will occur.
- using the VC++ compiler if the input consists of the name of a function-like macro, and that macro when invoked with no parameters returns a tuple, the macro erroneously returns 1, meaning that the input is empty.

For the macros which identify Boost PP data types the use of BOOST_VMD_IS_EMPTY occurs only after these macros recognize the beginning tuple in the input. So if the input begins with the situation above that causes BOOST_VMD_IS_EMPTY to work incorrectly, the Boost PP data type identifying macros will still work correctly. It is only if, when a beginning tuple has been found, that if the situation above occurs that induces the flaw with BOOST_VMD_IS_EMPTY with the rest of the input that we will have the same problem as BOOST_VMD_IS_EMPTY.

For the macros which identify v-identifiers and v-numbers the use of BOOST_VMD_IS_EMPTY occurs not only with the beginning input but with any input still remaining after the possible beginning v-identifier or v-number. So the same problem of BOOST_VMD_IS_EMPTY will occur with these identifying macros.

The obvious way to avoid the BOOST_VMD_IS_EMPTY problem wih the identifying macros is to design input so that the name of a function-like macro is never passed as a parameter. This can be done, if one uses VMD and does design situations where the input could contain a function-like macro name, by having that function-like macro name placed within a Boost PP data type, such as a tuple, without attempting to identify the type of the tuple element using VMD. In other word if the input is:

```
( SOME_FUNCTION_MACRO_NAME )
```

and we have the macro definition:

```
#define SOME_FUNCTION_MACRO_NAME(x,y) some_output
```

VMD can still parse the input as a tuple, if desired, using BOOST_VMD_IS_TUPLE without encountering the BOOST_VMD_IS_EMPTY problem. However if the input is:

```
SOME_FUNCTION_MACRO_NAME
```

either directly or through accessing the above tuple's first element, and the programmer attempts to use BOOST_VMD_IS_IDENTIFIER with this input, the BOOST_VMD_IS_EMPTY problem will occur.

Identifying macros and programming flexibility

The VMD identifying macros give the preprocessor metaprogrammer a great amount of flexibility when designing macros. It is not merely the flexibility of allowing direct parameters to a macro to be different data types, and having the macro work differently depending on the type of data passed to it, it is also the flexibility of allowing individual elements of the higher level Boost PP data types to be different data types and have the macro work correctly depending on the type of data type passed as part of those elements.

With this flexibility also comes a greater amount of responsibility. For the macro designer this responsibility is twofold:

- To carefully document the possible combinations of acceptable data and what they mean.

- To balance flexibility with ease of use so that the macro does not become so hard to understand that the programmer invoking the macro gives up using it intelligently.

For the programmer invoking a macro the responsibility is to understand the documentation and not attempt to pass to the macro data which may cause incorrect results or preprocessing errors.

Asserting and data types

The VMD macros for identifying v-types work best when the macro logic can take different paths depending on the type of data being passed for a macro parameter. But occasionally the preprocessor metaprogrammer wants to simply verify that the macro parameter data is of the correct v-type, else a preprocessor error should be generated to notify the programmer invoking the macro that the data passed is the incorrect type.

Using BOOST_VMD_ASSERT

The Boost PP library has a macro which produces a preprocessor error when the condition passed to it is 0. This macro is called BOOST_PP_ASSERT. The macro produces a preprocessor error by forcing a call to an internal macro with the wrong number of arguments. According to the C++ standard this should always cause an immediate preprocessor error for conforming compilers.

Unfortunately VC++ will only produce a warning when the wrong number of arguments are passed to a macro. Therefore the BOOST_PP_ASSERT macro does not produce a preprocessing error using VC++. Amazingly enough there appears to be no other way in which VC++ can be forced to issue a preprocessng error by invoking a macro (if you find one please tell me about it). However one can create invalid C++ as the output from a macro invocation which causes VC++ to produce a compiler error when the VC++ compiler later encounters the construct.

This is what the macro BOOST_VMD_ASSERT does. It takes the same conditional argument as BOOST_PP_ASSERT and it calls BOOST_PP_ASSERT when not used with VC++, otherwise if the condition is 0 it generates a compiler error by generating invalid C++ when used with VC++. The compiler error is generated by producing invalid C++ whose form is:

```
typedef char BOOST_VMD_ASSERT_ERROR[-1];
```

By passing a second optional argument, whose form is a preprocessing identifier, to BOOST_VMD_ASSERT you can generate the invalid C++ for VC++, if the first argument is 0, of the form:

```
typedef char optional_argument[-1];
```

instead. This may give a little more clarity, if desired, to the C++ error generated.

If the first conditional argument is not 0, BOOST_VMD_ASSERT produces no output.

BOOST_VMD_ASSERT Usage

To use the BOOST_VMD_ASSERT macro either include the general header:

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

or include the specific header:

```
#include <boost/vmd/assert.hpp>
```

Assertions for VMD data types

The v-types have their own assertion macros. These are largely just shortcuts for passing the result of the identifying macros to BOOST_VMD_ASSERT. These assertion macros are:

- emptiness, BOOST_VMD_ASSERT_IS_EMPTY
- v-identifier, BOOST_VMD_ASSERT_IS_IDENTIFIER
- v-number, BOOST_VMD_ASSERT_IS_NUMBER

- array, BOOST_VMD_ASSERT_IS_ARRAY
- list, BOOST_VMD_ASSERT_IS_LIST
- seq, BOOST_VMD_ASSERT_IS_SEQ
- tuple, BOOST_VMD_ASSERT_IS_TUPLE

Each of these macros take as parameters the exact same argument as their corresponding identifying macros. But instead of returning non-zero or 0, each of these macros produce a compiler error if the type of the input is not correct.

Each of these macros only check for its assertion when the macro BOOST_VMD_ASSERT_DATA is set to 1. By default BOOST_VMD_ASSERT_DATA is only set to 1 in compiler debug mode. The programmer can manually set BOOST_VMD_ASSERT_DATA to 1 prior to using one the data types assert macros if he wishes.

BOOST_VMD_ASSERT_... Usage

To use the individual BOOST_VMD_ASSERT_... macros either include the general header:

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

or include the specific header:

```
#include <boost/vmd/is_empty.hpp> // BOOST_VMD_ASSERT_IS_EMPTY
#include <boost/vmd/is_identifier.hpp> // BOOST_VMD_ASSERT_IS_IDENTIFIER
#include <boost/vmd/is_number.hpp> // BOOST_VMD_ASSERT_IS_NUMBER
#include <boost/vmd/array.hpp> // BOOST_VMD_ASSERT_IS_ARRAY
#include <boost/vmd/list.hpp> // BOOST_VMD_ASSERT_IS_LIST
#include <boost/vmd/seq.hpp> // BOOST_VMD_ASSERT_IS_SEQ
#include <boost/vmd/tuple.hpp> // BOOST_VMD_ASSERT_IS_TUPLE
```

Assertions and VC++

The VC++ compiler has a quirk when dealing with BOOST_VMD_ASSERT and the v-type assert macros. If you invoke one of the assert macros within another macro which would normally generate output preprocessor tokens, it is necessary when using VC++ to concatenate the result of the assert macro to whatever other preprocessor data is being generated, even if the assert macro does not generate an error.

As a simple example let us suppose we have a macro expecting a tuple and generating 1 if the tuple has more than 2 elements, otherwise it generates 0. Ordinarily we could write:

```
#define AMACRO(atuple) \
    BOOST_VMD_ASSERT_IS_TUPLE(atuple) \
    BOOST_PP_IIF(BOOST_PP_GREATER(BOOST_PP_TUPLE_SIZE(atuple), 2),1,0)
```

but for VC++ we must write

```
#define AMACRO(atuple) \
    BOOST_PP_CAT \
       ( \
       BOOST_VMD_ASSERT_IS_TUPLE(atuple), \
       BOOST_PP_IIF(BOOST_PP_GREATER(BOOST_PP_TUPLE_SIZE(atuple), 2),1,0) \
       )
```

VC++ does not work correctly in the first instance, erroneously getting confused as far as compiler ouput is concerned. But by using BOOST_PP_CAT in the second condition VC++ will work correctly with VMD assertions.

Parsing v-type sequences

Macro parameter syntax

In the normal use of Boost PP, data is passed as arguments to a macro in discrete units so that each parameter expects a single data type. A typical macro might be:

```
#define AMACRO(anumber,atuple,anidentifier) someoutput
```

where the 'atuple', having the form of (data1, data2, data3), itself may contain different data types of elements.

This is the standard macro design and internally it is the easiest way to pass macro data back and forth. The Boost PP library has a rich set of functionality to deal with all of its high-level data types and variadic data also offers another alternative to representing data with its own simpler functionality.

Occasionally designers of macros, especially for the use of others programmers within a particular library, have expressed the need for a macro parameter to allow a more C/C++ like syntax where a single parameter might mimic a C++ function-call or a C-like type modification syntax, or some other more complicated construct. Something along the lines of:

```
areturn afunction ( aparameter1, aparameter2, aparameter3 )
```

or

```
( type ) data
```

etc. etc.

In other words, from a syntactical level when designing possible macro input, is it possible to design parameter data to look more like C/C++ when macros are used in a library and still do a certain amount of preprocessor metaprogramming with such mixed token input ?

VMD has functionality which allows more than one type of preprocessing token, excluding an 'empty' token which always refers to entire input, to be part of a single parameter of input data, as long as all the top-level data of a parameter is of different v-types. What this means is that if some input consists of a sequence of v-types it is possible to extract the data for each v-type in the sequence.

In practicality what this means is that, given the examples just above, if 'areturn', 'afunction', and 'data' are v-identifiers it would be possible to parse either of the two inputs above so that one could identify the different v-types involved and do preprocessor meta-programming based on those results.

Before I give an explanation of how this is done using VMD functionality I would like to make two points:

- I am offering the possibility of more various syntaxes for the preprocessor metaprogrammer but I am aware that this can be easily abused. In general keeping things simple is usually better than making things overly complicated when it comes to the syntactical side of things in a computer language. A macro parameter syntactical possibility has to be understandable to be used.
- Using VMD to parse the individual types of a single parameter takes more preprocessing time than functionality offered with Boost PP data types, because a good deal of it is based on preprocessor recursion techniques.

Using VMD to parse input

We have seen how VMD can test input to a macro for a particular v-type. In these cases the input must be entirely of that particular type in order to return non-zero as opposed to 0.

VMD also has functionality to extract from input a particular v-type at the beginning of the input along with the remaining preprocessing tokens of that input. What this means is that given some input you can, as an example, extract the beginning tuple as well as the rest of the input. By recursively applying this technique to the rest of the input you can parse the input for its top-level data.

The one constraint using this technique is that the top-level input must consist of v-types, in other words preprocessor tokens which VMD understands. Therefore if your data is one of the examples above, you will be successful in using VMD. However if your preprocessor data takes the form of:

```
&name identifier ( param )
```

or

```
identifier "string literal"
```

or

```
identifier + number
```

or

```
identifier += 4.3
```

etc. etc.

you will not be able to parse the data using VMD since '&', "string literal", '+', '+=', and "4.3" are preprocessor tokens which are not top-level v-types and therefore VMD cannot handle them at the parsing level. You can still of course pass such data as preprocessing input to macros but you cannot use VMD to recognize the parts of such data.

This is similar to the fact that VMD cannot tell you what type preprocessor data is as a whole, using a VMD identifying macro, if the type is not one that VMD can handle.

On the other hand you can still use VMD to parse such tokens in the input if you use Boost PP data types as top-level v-types to do so. Such as:

```
( &name ) identifier ( param )
```

or

```
identifier ( "string literal" )
```

or

```
identifier ( + ) number
```

or

```
identifier ( += ) 4 ( . ) 3
```

I will be calling such input data, which consists of more than one v-type, by the term of a 'v-sequence'.

The succeeding topics explain the VMD functionality for parsing a v-sequence for each individual v-type.

Parsing v-identifiers

The macro BOOST_VMD_IDENTIFIER looks for a v-identifier at the beginning of a v-sequence. Just like BOOST_VMD_IS_IDENTIFIER it takes as mandatory parameters a v-sequence as the first parameter and a v-key as the second parameter. It returns a tuple of two elements. The first tuple element is the same return value as BOOST_VMD_IS_IDENTIFIER; an index, starting with 1, of the v-identifier corresponding to its v-key or 0 if no v-identifier is found. The second tuple element is

the rest of the v-sequence if a v-identifier is found or an empty element if a v-identifier is not found. Since a v-identifier can be found and the rest of the v-sequence can be empty, one always needs to check the first tuple element to determine if the v-identifier was found at the beginning of the v-sequence or not.

Usage

To use the BOOST_VMD_IDENTIFIER macro either include the general header:

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

or include the specific header:

```
#include <boost/vmd/identifier.hpp>
```

Simple v-sequences

A beginning v-identifier in a v-sequence can always be parsed using the mandatory parameters if the v-identifier is followed by:

- tuple ...
- v-number tuple ...
- v-number end-of_sequence
- end-of-sequence

A 'tuple' here refers to any pair of parenthesis, such as an array, a list, or the beginning of a seq.

As we will see more complicated v-sequences beginning with a v-identifier can be parsed using the optional parameters of BOOST_VMD_IDENTIFIER. These optional parameters will be discussed later.

Some examples:

```
#define BOOST_VMD_MAP_MYLIB_KEY1_CIRCLE
#define BOOST_VMD_MAP_MYLIB_KEY2_SQUARE
#define BOOST_VMD_MAP_MYLIB_KEY3_RECTANGLE
#define BOOST_VMD_MAP_MYLIB_KEY4_TRIANGLE
#define BOOST_VMD_MAP_MYLIB_KEY5_SHAPE

#define SOME_INPUT1 CIRCLE (atuple_elem1,atuple_elem2) any_vtypes

BOOST_VMD_IDENTIFIER
  (
  SOME_INPUT1 /* v-sequence */,
  MYLIB_KEY1_ /* CIRCLE, key which matches */
  )
```

expands to (1,(atuple_elem1,atuple_elem2) any_vtypes)

```
#define SOME_INPUT2 SQUARE 147 (atuple_elem1,atuple_elem2) any_vtypes

BOOST_VMD_IDENTIFIER
  (
  SOME_INPUT2 /* v-sequence */,
  (MYLIB_KEY1_ /* CIRCLE */,MYLIB_KEY2_ /* SQUARE, key which matches */)
  )
```

expands to (2,147 (atuple_elem1,atuple_elem2) any_vtypes)

```
#define SOME_INPUT3 RECTANGLE 33

BOOST_VMD_IDENTIFIER
  (
  SOME_INPUT3 /* v-sequence */,
  MYLIB_KEY3_ /* RECTANGLE, key which matches */
  )
```

expands to (1,33)

```
#define SOME_INPUT4 TRIANGLE

BOOST_VMD_IDENTIFIER
  (
  SOME_INPUT4 /* v-sequence */,
  (MYLIB_KEY1_ /* CIRCLE */,MYLIB_KEY2_ /* SQUARE */,MYLIB_KEY3_ /* TRIANGLE, key which matches */)
  )
```

expands to (3,)

```
#define SOME_INPUT5 AVIDENT any_vtypes

BOOST_VMD_IDENTIFIER
  (
  SOME_INPUT5 /* v-sequence */,
  MYLIB_KEY5_ /* SHAPE, key does not match */
  )
```

expands to (0,) because the v-identifier was not found at the beginning of the input.

Parsing v-identifiers using optional parameters

While parsing an v-sequence starting with a v-identifier can always be done for the fairly simple v-sequences situations previously mentioned, it is more challenging to do so when the v-identifier is followed by one or more other v-identifiers, or more than one v-number, or a combination of the two possible v-sequences. To do this one needs to use optional parameters to the BOOST_VMD_IDENTIFIER macro.

Recall that the BOOST_VMD_IDENTIFIER macro takes two mandatory parameters, the v-sequence and the v-key for the beginning identifier being parsed. The BOOST_VMD_IDENTIFIER macro also takes optional parameters to parse the more difficult sequences involving further v-identifiers or v-numbers.

Complicated v-sequences

Before I explain the optional parameters here are the forms of the more difficult v-sequences which BOOST_VMD_IDENTIFIER can parse with their help:

```
v-identifier v-identifier ... v-number-or-tuple-or-end-of-input
v-identifier v-number v-number ... tuple-or-end-of-input
v-identifier v-identifier ... v-number v-number ...tuple-or-end-of-input
```

These three supported situations allow * a v-identifier to be followed by from one to four additional v-identifiers followed by either a single v-number (followed by a tuple or end-of-sequence), tuple, or end-of-sequence. * a v-identifier to be followed by from two to five v-numbers before encountering either a tuple or end-of-sequence. * a v-identifier to be followed by from one to four additional v-identifiers, followed by from two to five v-numbers before encountering a tuple or end-of-sequence.

Parsing input sequences beginning with v-identifier(s), immediately followed by v-number(s), immediately followed by further v-identifier(s) are not supported. But of course this can be modified by having v-identifier(s), immediately followed followed by v-number(s), immediately followed followed by a tuple, followed by further v-identifier(s), which is supported in the third case above.

I am not necessarily advocating using such complicated sequences. It is more likely that a better design may have v-identifiers or v-numbers be separate elements of a tuple rather than consecutive elements of some v-sequence. While a reasonable case can be made for v-sequences of consecutive v-identifiers, since C++ mimics such syntax in a number of cases, certainly consecutive v-numbers would normally better be served in a tuple or seq or as variadic data rather than in a macro v-sequence, since the syntax of nnn nnn nnn nnn ... is not naturally seen in standard C++ as opposed to nnn,nnn,nnn,nnn ... .

Using the optional 3rd parameter

In order to find a v-identifier at the beginning of a v-sequence when it could be followed by a maximum of four subsequent v-identifiers, the optional 3rd parameter to BOOST_VMD_IDENTIFIER must be used. This is our v-sequence of:

```
v-identifier v-identifier ... v-number-or-tuple-or-end-of-input
```

This optional 3rd parameter is a seq, where each element is a v-key to possible subsequent v-identifiers. You can specify more seq elements than are actually found in the input v-sequence as long as the seq elements specified match v-identifiers that are found.

Some examples:

```
#define BOOST_VMD_MAP_MYLIB_KEY6_APPLE
#define BOOST_VMD_MAP_MYLIB_KEY7_BANANA
#define BOOST_VMD_MAP_MYLIB_KEY8_CHERRY
#define BOOST_VMD_MAP_MYLIB_KEY9_ORANGE
#define BOOST_VMD_MAP_MYLIB_KEY10_PEACH
#define BOOST_VMD_MAP_MYLIB_KEY11_PEAR

#define SOME_INPUT6 CHERRY BANANA PEACH (telem1,telem2)

BOOST_VMD_IDENTIFIER
  (
  SOME_INPUT6 /* v-sequence */,
  MYLIB_KEY8_ /* CHERRY, key which matches */,
  /* The optional 3rd parameter as a seq */
  ((MYLIB_KEY11_ /* PEAR */,MYLIB_KEY7_ /* BANANA, which matches */))
  (MYLIB_KEY10_ /* PEACH, which matches */)
  ((MYLIB_KEY6_ /* APPLE */,MYLIB_KEY9_ /* ORANGE */))
  /* This last seq element is extraneous for the particular input
     because the first two seq elements cover the two subsequent
     v-identifiers. But this shows you can have extra seq elements
     without affecting the ability to match */
  )
```

expands to (1,BANANA PEACH (telem1,telem2))

```
#define SOME_INPUT7 PEAR ORANGE APPLE PEACH 27

BOOST_VMD_IDENTIFIER
  (
  SOME_INPUT7 /* v-sequence */,
  (MYLIB_KEY6_ /* APPLE */,MYLIB_KEY11_ /* PEAR, key which matches */),
  /* The optional 3rd parameter as a seq */
  (MYLIB_KEY9_ /* ORANGE, which matches */)
  ((MYLIB_KEY7_ /* BANANA */,MYLIB_KEY6_ /* APPLE, which matches */))
  /* No subsequent seq element to match PEACH so beginning v-identifier
     is not found */
  )
```

expands to (0,)

The optional 3rd parameter is also used in order to find a v-identifier at the beginning of a v-sequence when it is followed by more than one v-number before a tuple or end-of-sequence is encountered. This is our v-sequence of:

```
v-identifier v-number v-number ... tuple-or-end-of-input
```

In this case the optional 3rd parameter is just a number, with a maximum of five, which specifies the maximum number of v-numbers which can follow the beginning identifier. You can specify a number which is greater than the actual number of v-numbers which follow the beginning v-identifier and the parsing will still succeed as long as the beginning v-identifier is found. But if you specify a number that is less than the number of subsequent v-numbers, the parsing will fail. You never need to specify the optional 3rd parameter as a number if the v-identifier in the beginning of a v-sequence is followed by a single v-number, but you are allowed to do so.

Some examples, using our key-identifier macros above:

```
#define SOME_INPUT8 APPLE 37 142 (telem1,telem2)

BOOST_VMD_IDENTIFIER
  (
  SOME_INPUT8 /* v-sequence */,
  ( MYLIB_KEY10_ /* PEACH */, MYLIB_KEY6_ /* APPLE, key which matches */ ),
  /* The optional 3rd parameter as a number */
  3 /* succeeds because there are less than or equal to 3 following v-numbers */
  )
```

expands to (2,37 142 (telem1,telem2))

```
#define SOME_INPUT9 ORANGE 25 99 234 56

BOOST_VMD_IDENTIFIER
  (
  SOME_INPUT9 /* v-sequence */,
  MYLIB_KEY9_ /* ORANGE, key which matches */,
  /* The optional 3rd parameter as a number */
  2 /* fails because there are more than 2 following v-numbers */
  )
```

expands to (0,)

```
#define SOME_INPUT10 BANANA 226

BOOST_VMD_IDENTIFIER
  (
  SOME_INPUT10 /* v-sequence */,
  MYLIB_KEY7_ /* BANANA, key which matches */,
  /* The optional 3rd parameter as a number */
  1 /* Not needed but it does no harm */
  )
```

expands to (1,226)

Using the optional 4th parameter

The optional 4th parameter is used in order to find a v-identifier at the beginning of the v-sequence when it is followed by v-identifiers and more than one v-number before a tuple or end-of-sequence. This is our v-sequence of:

```
v-identifier v-identifier ... v-number v-number ...tuple-or-end-of-input
```

In this situation the optional 3rd parameter is the seq we used for subsequent v-identifiers and the optional 4th parameter is the number we used for subsequent v-numbers. When we use both optional parameters we may still succeed if we encounter a single v-number (followed by a tuple or end-of-sequence), tuple, or end-of-sequence before the seq matches are exhausted or, once we start matching v-numbers, if we encounter a tuple or end-of-sequnce before the number matches are exhausted. But once we encounter consecutive v-numbers in the v-sequence, we need to have matched exactly the number of seqs specified.

Some examples, using our key-identifier macros above:

```
#define SOME_INPUT11 CHERRY BANANA 25 99 234 (telem1,telem2)

BOOST_VMD_IDENTIFIER
  (
  SOME_INPUT11 /* v-sequence */,
  MYLIB_KEY8_ /* CHERRY, key which matches */,
  ((MYLIB_KEY11_ /* PEAR */,MYLIB_KEY7_ /* BANANA, which matches */)),
  4 /* succeeds because there are less than or equal to 4 following v-numbers */
  )
```

expands to (1,BANANA 25 99 234 (telem1,telem2))

```
#define SOME_INPUT12 PEAR ORANGE 8

BOOST_VMD_IDENTIFIER
  (
  SOME_INPUT12 /* v-sequence */,
  (MYLIB_KEY11_ /* PEAR, key which matches */, MYLIB_KEY8_ /* CHERRY*/),
  ((MYLIB_KEY7_ /* BANANA */,MYLIB_KEY9_ /* ORANGE, which matches */))
  ((MYLIB_KEY9_ /* ORANGE */,MYLIB_KEY8_ /* CHERRY */)),
  /* This last seq element is extraneous for the particular input
     because the first seq element covers the subsequent
     v-identifier and there is at most one v-number. This shows you
     can have extra seq elements without affecting the ability to match
     as long as you have no following consecutive v-numbers. */
  2
  /* succeeds because there are less than or equal to 2 following v-numbers
     but is not needed for the particular input because only a single
     v-number occurs */
  )
```

expands to (1,ORANGE)

```
#define SOME_INPUT13 APPLE PEACH PEAR 37 42

BOOST_VMD_IDENTIFIER
  (
  SOME_INPUT13 /* v-sequence */,
  MYLIB_KEY6_ /* APPLE, key which matches */,
  ((MYLIB_KEY7_ /* BANANA */,MYLIB_KEY9_ /* ORANGE */,MYLIB_KEY10_ /* PEACH, which matches*/))
  (MYLIB_KEY11_ /* PEAR, which matches */)
  (MYLIB_KEY8_ /* CHERRY,
    this negates the match since consecutive v-numbers occur
    next in the v-sequence and the 4th parameter is specified.
    If the 4th parameter had not been specified then this seq
    element would have been extraneous but would not have negated
    the match. */),
  2
  )
```

expands to (0,)

```
#define SOME_INPUT14 BANANA CHERRY 142 50

BOOST_VMD_IDENTIFIER
  (
  SOME_INPUT14 /* v-sequence */,
  MYLIB_KEY7_ /* BANANA, key which matches */,
  ((MYLIB_KEY7_ /* BANANA */,MYLIB_KEY8_ /* CHERRY, which matches */,MYLIB_KEY10_ /* PEACH */)),
  3 /* succeeds because there are less than or equal to 3 following v-numbers */
  )
```

expands to (1,CHERRY 142 50)

V-identifiers helper macros

A number of macros are helper versions of BOOST_VMD_IDENTIFIER, having the same two mandatory parameters and two optional parameters which BOOST_VMD_IDENTIFIER takes. All these macros work, just like BOOST_VMD_IDENTIFIER, to parse a v-identifier occuring at the beginning of a v-sequence.

These macros are:

BOOST_VMD_BEGIN_IDENTIFIER: expands to just the index, starting with 1, of the beginning identifier found, or 0 if no beginning identifier has been found.

BOOST_VMD_AFTER_IDENTIFIER: expands to the preprocessor tokens after the beginning identifier. If the identifier is not found, expands to nothing.

BOOST_VMD_IS_BEGIN_IDENTIFIER: expands to 1 if input begins with an identifier, else expands to 0 if it does not.

Usage

All of these helper macros are in the same header file as the BOOST_VMD_IDENTIFIER macro.

You can use the general header file:

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

or include the specific header:

```
#include <boost/vmd/identifier.hpp>
```

Parsing v-numbers

The macro BOOST_VMD_NUMBER looks for a v-number at the beginning of a v-sequence. Just like BOOST_VMD_IS_NUMBER it takes as a mandatory parameter a v-sequence as the first parameter. It returns a tuple of two elements. The first tuple element is the number if a v-number has been found or is an empty element if a v-number has not been found. The second tuple element is the rest of the v-sequence if a v-number has been found or an empty element if no v-number has been found. Since a v-number can be found and the rest of the v-sequence can be empty, one always needs to check the first tuple element to determine if the v-number was found at the beginning of the v-sequence or not.

Usage

To use the BOOST_VMD_NUMBER macro either include the general header:

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

or include the specific header:

```
#include <boost/vmd/number.hpp>
```

Simple v-sequences

A beginning v-number in a v-sequence can always be parsed using just the mandatory v-sequence parameter if the v-number is followed by:

- tuple ...
- end-of-sequence

A 'tuple' here refers to any pair of parenthesis, such as an array, a list, or the beginning of a seq.

As we will see more complicated v-sequences beginning with a v-number can be parsed using the optional parameters of BOOST_VMD_NUMBER. These optional parameters will be discussed later.

Some examples:

```
#define SOME_INPUT_N1 234 (tupe1,tupe2)

BOOST_VMD_NUMBER(SOME_INPUT_N1)
```

expands to (234,(tupe1,tupe2))

```
#define SOME_INPUT_N2 147

BOOST_VMD_NUMBER(SOME_INPUT_N2)
```

expands to (147,)

```
#define SOME_INPUT_N3 5 (seq1)(seq2) 47

BOOST_VMD_NUMBER(SOME_INPUT_N3)
```

expands to (5,(seq1)(seq2) 47)

```
#define SOME_INPUT_N4 name 227

BOOST_VMD_NUMBER(SOME_INPUT_N4)
```

expands to (,) since the v-sequence does not beginning with a v-number

```
#define SOME_INPUT_N5 79 120 (tupe1,tupe2)
#define SOME_INPUT_N6 79 name (tupe1,tupe2)

BOOST_VMD_NUMBER(SOME_INPUT_N5)
BOOST_VMD_NUMBER(SOME_INPUT_N6)
```

either one expands to (,) since the beginning number is not followed by a tuple or end-of-sequence.

See the next topic for how to parse this last example using the optional parameters of BOOST_VMD_NUMBER.

Parsing v-numbers using optional parameters

While parsing a v-sequence starting with a v-number can always be done for the fairly simple v-sequences situations previously mentioned, it is more challenging to do so when the v-number is followed by one or more other v-numbers, or one or more v-identi-

fiers, or a combination of the two possible v-sequences. To do this one needs to use optional parameters to the BOOST_VMD_NUMBER macro.

Recall that the BOOST_VMD_NUMBER macro takes a single mandatory parameter, the v-sequence. The BOOST_VMD_NUMBER macro also takes optional parameters to parse the more difficult sequences involving further v-numbers or v-identifiers.

Complicated v-sequences

Before I explain the optional parameters here are the forms of the more difficult v-sequences which BOOST_VMD_NUMBER can parse with their help:

```
v-number v-numbers ... tuple-or-end-of-input
v-number v-identifiers ... tuple-or-end-of-input
v-number v-numbers ... v-identifiers ... tuple-or-end-of-input
```

These three supported situations also allow * a v-number to be followed by from one to four additional v-numbers followed by a tuple, or end-of-sequence. * a v-number to be followed by from one to five v-identifiers followed by a tuple or end-of-sequence. * a v-number to be followed by from one to four additional v-numbers, followed by from one to five v-identifiers before encountering a tuple or end-of-sequence.

Parsing input sequences beginning with v-number(s), immediately followed by v-identifier(s), immediately followed by further v-number(s) are not supported. But of course this can be modified by having v-number(s), immediately followed followed by v-identifier(s), immediately followed followed by a tuple, followed by further v-number(s), which is supported by the third case above.

Using the optional 2nd parameter

In order to find a v-number at the beginning of a v-sequence when it could be followed by a maximum of four subsequent v-numbers, the optional 2nd parameter to BOOST_VMD_NUMBER must be used. This is our v-sequence of:

```
v-number v-numbers ... tuple-or-end-of-input
```

In this case the optional 2nd parameter is just a number, with a maximum of four, which specifies the maximum number of subsequent v-numbers which can follow the beginning v-number. You can specify a number which is greater than the actual number of v-numbers which follow the beginning v-identifier and the parsing will still succeed as long as the beginning v-number is found. But if you specify a number that is less than the number of subsequent v-numbers, the parsing will fail. You never need to specify the optional 2nd parameter as a number if the v-number in the beginning of a v-sequence is followed by a tuple or end-of-input, but you are allowed to do so.

Some examples:

```
#define SOME_INPUT_N7 37 142 (telem1,telem2)

BOOST_VMD_NUMBER
  (
  SOME_INPUT_N7 /* v-sequence */,
  /* The optional 2nd parameter as a number */
  3 /* succeeds because there are less than or equal to 3 following v-numbers */
  )
```

expands to (37,142 (telem1,telem2))

```
#define SOME_INPUT_N8 25 99 234 56

BOOST_VMD_NUMBER
  (
  SOME_INPUT_N8 /* v-sequence */,
  /* The optional 2nd parameter as a number */
  2 /* fails because there are more than 2 following v-numbers */
  )
```

expands to (,)

```
#define SOME_INPUT_N9 226

BOOST_VMD_NUMBER
  (
  SOME_INPUT_N9 /* v-sequence */,
  /* The optional 2nd parameter as a number */
  1 /* Not needed but it does no harm */
  )
```

expands to (226,)

The optional 2nd parameter is also used in order to find a v-number at the beginning of a v-sequence when it is followed by one or more v-identifiers before a tuple or end-of-sequence is encountered. This is our v-sequence of:

```
v-number v-identifiers ... tuple-or-end-of-input
```

This optional 2nd parameter is a seq, where each element is a v-key to possible subsequent v-identifiers. You can specify more seq elements than are actually found in the v-sequence as long as the seq elements specified match v-identifiers that are found.

Some examples:

```
#define BOOST_VMD_MAP_MYLIB_KEY6_APPLE
#define BOOST_VMD_MAP_MYLIB_KEY7_BANANA
#define BOOST_VMD_MAP_MYLIB_KEY8_CHERRY
#define BOOST_VMD_MAP_MYLIB_KEY9_ORANGE
#define BOOST_VMD_MAP_MYLIB_KEY10_PEACH
#define BOOST_VMD_MAP_MYLIB_KEY11_PEAR

#define SOME_INPUT_N10 165 CHERRY BANANA PEACH (telem1,telem2)

BOOST_VMD_NUMBER
  (
  SOME_INPUT_N10 /* v-sequence */,
  /* The optional 2nd parameter as a seq */
  ((MYLIB_KEY8_ /* CHERRY, which matches */,MYLIB_KEY7_ /* BANANA */))
  ((MYLIB_KEY11_ /* PEAR */,MYLIB_KEY7_ /* BANANA, which matches */))
  (MYLIB_KEY10_ /* PEACH, which matches */)
  ((MYLIB_KEY6_ /* APPLE */,MYLIB_KEY9_ /* ORANGE */))
  /* This last seq element is extraneous for the particular input
     because the first three seq elements cover the three
     v-identifiers. But this shows you can have extra seq elements
     without affecting the ability to match */
  )
```

expands to (165,CHERRY BANANA PEACH (telem1,telem2))

```
#define SOME_INPUT_N11 27 PEAR ORANGE APPLE PEACH

BOOST_VMD_NUMBER
  (
  SOME_INPUT_N11 /* v-sequence */,
  /* The optional 2nd parameter as a seq */
  ((MYLIB_KEY7_ /* BANANA */,MYLIB_KEY11_ /* PEAR, which matches */))
  (MYLIB_KEY9_ /* ORANGE, which matches */)
  ((MYLIB_KEY7_ /* BANANA */,MYLIB_KEY6_ /* APPLE, which matches */))
  /* No subsequent seq element to match PEACH so beginning v-identifier
     is not found */
  )
```

expands to (,)

Using the optional 3rd parameter

The optional 3rd parameter is used in order to find a v-number at the beginning of the v-sequence when it is followed by one to four v-numbers and one to five v-identifiers before a tuple or end-of-sequence. This is our v-sequence of:

```
v-number v-numbers ... v-identifiers ... tuple-or-end-of-input
```

In this situation the optional 2nd parameter is the number we used for subsequent v-numbers and the optional 3rd parameter is the seq we used for subsequent v-identifiers. When we use both optional parameters we may still succeed if we encounter a tuple or end-of-sequence before the v-number matches are exhausted or, once we start matching v-identifiers, if we encounter a tuple or end-of-sequence before the v-identifiers matches are exhausted. But once we encounter v-identifiers in the v-sequence, we need to have matched exactly the number of v-numbers specified.

Some examples, using our key-identifier macros above:

```
#define SOME_INPUT_N12 25 99 234 CHERRY BANANA (telem1,telem2)

BOOST_VMD_NUMBER
  (
  SOME_INPUT_N12 /* v-sequence */,
  2, which matches the 2 subsequent v-numbers
  (MYLIB_KEY8_ /* CHERRY, which matches */)
  ((MYLIB_KEY11_ /* PEAR */,MYLIB_KEY7_ /* BANANA, which matches */))
  )
```

expands to (25,99 234 CHERRY BANANA (telem1,telem2))

```
#define SOME_INPUT_N13 8 253 PEAR ORANGE

BOOST_VMD_NUMBER
  (
  SOME_INPUT_N13 /* v-sequence */,
  1, which matches the 1 subsequent v-number
  ((MYLIB_KEY11_ /* PEAR, which matches */, MYLIB_KEY8_ /* CHERRY*/))
  ((MYLIB_KEY7_ /* BANANA */,MYLIB_KEY9_ /* ORANGE, which matches */))
  ((MYLIB_KEY9_ /* ORANGE */,MYLIB_KEY8_ /* CHERRY */))
  /* This last seq element is extraneous for the particular input
     because the first two seq element covers the subsequent
     v-identifiers. This shows you can have extra seq elements without
     affecting the ability to match as long as you have the correct number
     of subsequent v-numbers specified. */
  )
```

expands to (8,253 PEAR ORANGE)

```
#define SOME_INPUT_N14 37 42 156 254

BOOST_VMD_NUMBER
  (
  SOME_INPUT_N14 /* v-sequence */,
  4, /* Succeeds because he number is greater or equal to the amount of
        subsequent v-numbers and there are no subsequent v-identifiers
        in the input before a tuple or end-of-sequence is encoumntered */
  /* The 3rd parameter seq is extraneous for the given input because there
     are no subsequent v-identifiers to match, but do not negate the
     match because it i specified */
  ((MYLIB_KEY7_ /* BANANA */,MYLIB_KEY9_ /* ORANGE */,MYLIB_KEY10_ /* PEACH */))
  (MYLIB_KEY11_ /* PEAR, which matches */)
  (MYLIB_KEY8_ /* CHERRY */)
  )
```

expands to (37,42 156 254)

```
#define SOME_INPUT_N15 142 50 133 29 BANANA CHERRY

BOOST_VMD_NUMBER
  (
  SOME_INPUT_N15 /* v-sequence */,
  4, /* Fails because there must be exactly the correct number of subsequent
        v-numbers if there are subsequent v-identifiers */
  (MYLIB_KEY7_ /* BANANA, which matches */)
  ((MYLIB_KEY7_ /* BANANA */,MYLIB_KEY8_ /* CHERRY, which matches */,MYLIB_KEY10_ /* PEACH */))
  )
```

expands to (,)

V-number helper macros

A number of macros are helper versions of BOOST_VMD_NUMBER, having the same single mandatory parameter and two optional parameters which BOOST_VMD_NUMBER takes. All these macros work with a v-number occuring at the beginning of a v-sequence.

These macros are:

BOOST_VMD_BEGIN_NUMBER: expands to the beginning v-number or nothing if no beginning v-number has been found.

BOOST_VMD_AFTER_NUMBER: expands to the preprocessor tokens after the beginning v-number. If the v-number is not found, expands to nothing.

BOOST_VMD_IS_BEGIN_NUMBER: expands to 1 if input begins with a v-number, else expands to 0 if it does not.

Usage

All of these helper macros are in the same header file as the BOOST_VMD_NUMBER macro.

You can use the general header file:

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

or include the specific header:

```
#include <boost/vmd/number.hpp>
```

Parsing Boost PP data

There are macros for the Boost PP data types that can parse the particular data type at the beginning of a v-sequence. Unlike the equivalent functionality for v-identifiers and v-numbers the only parameter that is allowed is the v-sequence itself.

The macros are:

- BOOST_VMD_ARRAY for an array
- BOOST_VMD_LIST for a list
- BOOST_VMD_SEQ for a seq
- BOOST_VMD_TUPLE for a tuple

Each of these macros expand to a tuple of two elements. If the particular data type is found at the beginning of the v-sequence the first tuple element is that data type and the second tuple element is the rest of the input v-sequence. If the particular data type is not found at the beginning of the input v-sequence of both tuple elements is empty.

Usage

In order to use any of the macros you can include the general header:

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

or include the appropriate specific header:

```
#include <boost/vmd/array.hpp> // BOOST_VMD_ARRAY
#include <boost/vmd/list.hpp> // BOOST_VMD_LIST
#include <boost/vmd/seq.hpp> // BOOST_VMD_SEQ
#include <boost/vmd/tuple.hpp> // BOOST_VMD_TUPLE
```

Data types and tuples

An array and a non-empty list are also tuples. A seq is a series of single element tuples. Therefore if a v-sequence begins with an array, non-empty list, or seq, and you use BOOST_VMD_TUPLE passing that v-sequence, you will successfully find the beginning tuple. In the case of an array or a non-empty list what is returned will be exactly the same as a tuple. In the case of a seq what is returned will only be the same if the seq is a single one-element tuple.

As an example:

```
#define INPUT_D1 (3,(a,b,c)) data

BOOST_VMD_ARRAY(INPUT_D1)
BOOST_VMD_TUPLE(INPUT_D1)
```

expands to ((3,(a,b,c)),data)

```
#define INPUT_D2 (a,(b,(c,BOOST_PP_NIL))) 204

BOOST_VMD_LIST(INPUT_D2)
BOOST_VMD_TUPLE(INPUT_D2)
```

expands to ((a,(b,(c,BOOST_PP_NIL))),204)

```
#define INPUT_D3 (a)(b)(c) anything

BOOST_VMD_SEQ(INPUT_D3)
```

expands to ((a)(b)(c),anything)

```
BOOST_VMD_TUPLE(INPUT_D3)
```

expands to ((a),(b)(c) anything)

On the other hand not every tuple is a non-empty list or array while only a single element tuple is also a seq.

As an example:

```
#define INPUT_D4 (z,(a,b,c)) data

BOOST_VMD_ARRAY(INPUT_D4)
```

expands to (,) since the beginning tuple is not an array

```
BOOST_VMD_TUPLE(INPUT_D4)
```

expands to ((z,(a,b,c)),data) since there is a beginning tuple

```
#define INPUT_D5 (a,(b,(c))) 204

BOOST_VMD_LIST(INPUT_D5)
```

expands to (,) since the beginning tuple is not a list (missing BOOST_PP_NIL terminator)

```
BOOST_VMD_TUPLE(INPUT_D5)
```

expands to ((a,(b,(c))),204) since there is a beginning tuple

```
#define INPUT_D6 (a,b)(c) anything

BOOST_VMD_SEQ(INPUT_D6)
```

expands to (,) since the beginning tuple here cannot start a seq

```
BOOST_VMD_TUPLE(INPUT_D6)
```

expands to ((a,b),(c) anything) since there is a beginning tuple

```
#define INPUT_D7 (abc) anything

BOOST_VMD_SEQ(INPUT_D7)
BOOST_VMD_TUPLE(INPUT_D7)
```

expands to ((abc),anything) since the beginning tuple is also a beginning seq

PP-data helper macros

A number of macros are helper versions of each of the parsing Boost PP data type macros. All of them take as a parameter the v-sequence to parse.

These macros are:

BOOST_VMD_BEGIN_ARRAY: BOOST_VMD_BEGIN_LIST: BOOST_VMD_BEGIN_SEQ: BOOST_VMD_BEGIN_TUPLE:

expands to the appropriate beginning Boost PP data type or nothing if the appropriate beginning Boost PP data type is not found.

BOOST_VMD_AFTER_ARRAY: BOOST_VMD_AFTER_LIST: BOOST_VMD_AFTER_SEQ: BOOST_VMD_AFTER_TUPLE:

expands to the preprocessor tokens after the beginning appropriate Boost PP data type. If the beginning Boost PP data type is not found, expands to nothing.

BOOST_VMD_IS_BEGIN_ARRAY: BOOST_VMD_IS_BEGIN_LIST: BOOST_VMD_IS_BEGIN_SEQ: BOOST_VMD_IS_BEGIN_TUPLE:

expands to 1 if input begins with the appropriate Boost PP data type, else expands to 0 if it does not.

Usage

In order to use any of the macros you can include the general header:

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

or include the appropriate specific header:

```
#include <boost/vmd/array.hpp>
#include <boost/vmd/list.hpp>
#include <boost/vmd/seq.hpp>
#include <boost/vmd/tuple.hpp>
```

Generating emptiness and identity

Using BOOST_PP_EMPTY and BOOST_PP_IDENTITY

Boost PP Has a macro called BOOST_PP_EMPTY() which expands to nothing.

Ordinarily this would not seem that useful, but the macro can be used in situations where one wants to return a specific value even though a further macro call syntax is required taking no parameters. This sort of usefulness occurs in Boost PP when there are two paths to take depending on the outcome of a BOOST_PP_IF or BOOST_PP_IIF logic. Here is an artificial example:

```
#define MACRO_CHOICE(parameter) \
    BOOST_PP_IIF(parameter) \
        ( \
        MACRO_CALL_IF_PARAMETER_1, \
        SOME_FIXED_VALUE BOOST_PP_EMPTY \
        ) \
    ()

#define MACRO_CALL_IF_PARAMETER_1() some_processing
```

In the general logic above is: if parameter is 1 another macro is invoked, whereas if the parameter is 0 some fixed value is returned. The reason that this is useful is that one may not want to code the MACRO_CHOICE macro in this way:

```
#define MACRO_CHOICE(parameter) \
    BOOST_PP_IIF(parameter) \
        ( \
        MACRO_CALL_IF_PARAMETER_1(), \
        SOME_FIXED_VALUE BOOST_PP_EMPTY() \
        )

#define MACRO_CALL_IF_PARAMETER_1() some_processing
```

because it is inefficient. The invocation of MACRO_CALL_IF_PARAMETER_1 will still be generated even when 'parameter' is 0.

This idiom of returning a fixed value through the use of BOOST_PP_EMPTY is so useful that Boost PP has an accompany macro to BOOST_PP_EMPTY to work with it. This accompanying macro is BOOST_PP_IDENTITY(value)(). Essentially BOOST_PP_IDENTITY returns its value when it is invoked. Again, like BOOST_PP_EMPTY, the final invocation must be done with no value.

Our example from above, which originally used BOOST_PP_EMPTY to return a fixed value, is now:

```
#define MACRO_CHOICE(parameter) \
    BOOST_PP_IIF(parameter) \
        ( \
        MACRO_CALL_IF_PARAMETER_1, \
        BOOST_PP_IDENTITY(SOME_FIXED_VALUE) \
        ) \
    ()

#define MACRO_CALL_IF_PARAMETER_1() some_processing
```

The macro BOOST_PP_IDENTITY is actually just:

```
#define BOOST_PP_IDENTITY(value) value BOOST_PP_EMPTY
```

so you can see how it is essentially a shorthand for the common case originally shown at the top of returning a value through the use of BOOST_PP_EMPTY.

Using BOOST_VMD_EMPTY and BOOST_VMD_IDENTITY

The one problem when using BOOST_PP_EMPTY and BOOST_PP_IDENTITY is that the final invocation must be with no parameters. This is very limiting. If the final invocation must be with one or more parameters you cannot use BOOST_PP_EMPTY or BOOST_PP_IDENTITY. In other words, making a change to either of our two examples:

```
#define MACRO_CHOICE(parameter1,parameter2) \
    BOOST_PP_IIF(parameter1) \
        ( \
        MACRO_CALL_IF_PARAMETER_1, \
        SOME_FIXED_VALUE BOOST_PP_EMPTY \
        ) \
    (parameter1,parameter2)

#define MACRO_CALL_IF_PARAMETER_1(parameter1,parameter2) some_processing_using_both_parameters
```

or

```
#define MACRO_CHOICE(parameter1,parameter2) \
    BOOST_PP_IIF(parameter1) \
        ( \
        MACRO_CALL_IF_PARAMETER_1, \
        BOOST_PP_IDENTITY(SOME_FIXED_VALUE) \
        ) \
    (parameter1,parameter2)

#define MACRO_CALL_IF_PARAMETER_1(parameter1,parameter2) some_processing_using_both_parameters
```

will produce a preprocessing error since the final invocation to either BOOST_PP_EMPTY or BOOST_PP_IDENTITY can not be done with 1 or more parameters.

It would be much more useful if the final invocation could be done with any nymber of parameters. This is where variadic macros solve the problem. The BOOST_VMD_EMPTY and BOOST_VMD_IDENTITY macros have the exact same functionality as their Boost PP counterparts but the final invocation can be made with any number of parameters, and those parameters are just ignored.

Now for our two examples we can have:

```
#define MACRO_CHOICE(parameter1,parameter2) \
    BOOST_PP_IIF(parameter1) \
        ( \
        MACRO_CALL_IF_PARAMETER_1, \
        SOME_FIXED_VALUE BOOST_VMD_EMPTY \
        ) \
    (parameter1,parameter2)

#define MACRO_CALL_IF_PARAMETER_1(parameter1,parameter2) some_processing_using_parameters
```

or

```
#define MACRO_CHOICE(parameter1,parameter2) \
    BOOST_PP_IIF(parameter1) \
        ( \
        MACRO_CALL_IF_PARAMETER_1, \
        BOOST_VMD_IDENTITY(SOME_FIXED_VALUE) \
        ) \
    (parameter1,parameter2)

#define MACRO_CALL_IF_PARAMETER_1(parameter1,parameter2) some_processing_using_parameters
```

and our macros will compile without preprocessing errors and work as expected. Both BOOST_VMD_EMPTY and BOOST_VMD_IDENTITY will take any number of parameters in their invocation, which makes them useful for a final invocation no matter what is being passed.

Usage for BOOST_VMD_EMPTY and BOOST_VMD_IDENTITY

To use the BOOST_VMD_EMPTY macro either include the general header:

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

or include the specific header:

```
#include <boost/vmd/empty.hpp>
```

To use the BOOST_VMD_IDENTITY macro either include the general header:

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

or include the specific header:

```
#include <boost/vmd/identity.hpp>
```

Using BOOST_VMD_EMPTY and BOOST_VMD_IDENTITY with VC++

Unfortunately the Visual C++ preprocessor has a problem when a macro expands to something followed by a variadic macro which expands to nothing. This is the case when using BOOST_VMD_EMPTY following some non-empty expansion, or the equivalent use of BOOST_VMD_IDENTITY. As strange as it sounds this VC++ preprocessor problem is solved by concatenating the result using BOOST_PP_CAT with an empty value. But then again the many non-standard behaviors of VC++ are difficult to understand or even track.

In order to make this technique transparent when used with a C++ standard conforming preprocessor or VC++ non-standard preprocessor you can use the BOOST_VMD_IDENTITY_RESULT macro passing to it a single parameter which is a result returned from a macro which uses BOOST_VMD_IDENTITY (or its equivalent 'value BOOST_VMD_EMPTY' usage).

Given our MACRO_CHOICE example above, if you have another macro invoking MACRO_CHOICE simply enclose that invocation within BOOST_VMD_IDENTITY_RESULT. As in the very simple:

```
#define CALLING_MACRO_CHOICE(parameter1,parameter2) \
    BOOST_VMD_IDENTITY_RESULT(MACRO_CHOICE(parameter1,parameter2))
```

Using BOOST_VMD_EMPTY and BOOST_VMD_IDENTITY in this way will ensure they can be used without preprocessing problems with either VC++ or any C++ standard conforming preprocessor.

Usage for BOOST_VMD_IDENTITY_RESULT

The macro BOOST_VMD_IDENTITY_RESULT is in the same header file as BOOST_VMD_IDENTITY, so to use the BOOST_VMD_IDENTITY_RESULT macro either include the general header:

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

or include the specific header:

```
#include <boost/vmd/identity.hpp>
```

Emptiness and Boost PP data types

While the VMD library supports the identification of empty input through the BOOST_VMD_IS_EMPTY macro, it is also possible to use Boost PP data types to pass empty data.

Empty arrays

The Boost PP array is largely obsolete when variadic macros are used, since the Boost PP tuple contains all of the functionality of the array, except in a single area. The array can truly be empty, and an empty array takes the form of:

```
(0,())
```

This is an empty array since its size is 0 and its data is empty. The VMD library has functionality to identify an empty array using the BOOST_VMD_IS_EMPTY_ARRAY macro. This macro takes a parameter as input and determines whether or not it is an empty array, expanding to 1 if it is and 0 if it is not.

Usage for BOOST_VMD_IS_EMPTY_ARRAY

To use the BOOST_VMD_IS_EMPTY_ARRAY macro either include the general header:

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

or include the specific header:

```
#include <boost/vmd/array.hpp>
```

Empty lists

A Boost PP list can be empty, and an empty list takes the form of:

```
BOOST_PP_NIL
```

This is a truly empty list, whose size is 0 and which contains no list data. The VMD library has functionality to identify an empty list using the BOOST_VMD_IS_EMPTY_LIST macro. This macro takes a parameter as input and determines whether or not it is an empty list, expanding to 1 if it is and 0 if it is not.

Usage for BOOST_VMD_IS_EMPTY_LIST

To use the BOOST_VMD_IS_EMPTY_LIST macro either include the general header:

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

or include the specific header:

```
#include <boost/vmd/list.hpp>
```

Emptiness with seqs and tuples

The Boost PP data types of a seq or a tuple are never empty. All seqs or tuples have a minimum size of 1. The form of:

```
()
```

can be either a seq whose single element contains no data or a tuple containing a single element of no data. In both cases the size of the seq or tuple are 1 even if the data is empty.

Even though a seq or a tuple is never empty you can choose to consider a seq or tuple whose single element data is empty as an empty construct. The VMD library has a macro called BOOST_VMD_IS_PARENS_EMPTY to identify this construct. The macro takes a single parameter and returns 1 if the parameter is a set of parenthesis with no data, or 0 if it is not.

Usage for BOOST_VMD_IS_PARENS_EMPTY

To use the BOOST_VMD_IS_PARENS_EMPTY macro either include the general header:

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

or include the specific header:

```
#include <boost/vmd/is_parens_empty.hpp>
```

Conversions of "empty" data

The Boost PP array and list can be empty but a seq or a tuple are never considered empty, even if we use the form of a seq or tuple which represents a set of parenthesis which contains no data. Because of this we can convert between an empty array or an empty list, but if we convert from an empty array or list to a seq or tuple we get undefined behavior. On the other side if we convert from a seq or a tuple which is represented by a set of parenthesis which contain no data, we end up with an array or list whose size is 1 but whose single element contains no data. The following table illustrated these conversions in Boost PP:

Table 1. Boost PP data conversions for "empty" data

Data type	To array	To list	To seq	To tuple
Array	N/A	BOOST_PP_ARRAY_TO_LIST result = empty list, BOOST_PP_NIL	BOOST_PP_ARRAY_TO_SEQ result = undefined behavior	BOOST_PP_ARRAY_TO_TUPLE result = undefined behavior
List	BOOST_PP_LIST_TO_ARRAY BOOST_PP_LIST_TO_ARRAY_D result = empty array, (0,())	N/A	BOOST_PP_LIST_TO_SEQ BOOST_PP_LIST_TO_SEQ_R result = undefined behavior	BOOST_PP_LIST_TO_TUPLE BOOST_PP_LIST_TO_TUPLE_R result = undefined behavior
Seq	BOOST_PP_SEQ_TO_ARRAY result = (1,())	BOOST_PP_SEQ_TO_LIST(result = ((),BOOST_PP_NIL)	N/A	BOOST_PP_SEQ_TO_TUPLE result = empty parenthesis, ()
Tuple	BOOST_PP_TUPLE_TO_ARRAY result = (1,())	BOOST_PP_TUPLE_TO_LIST result = ((),BOOST_PP_NIL)	BOOST_PP_TUPLE_TO_SEQ result = empty parenthesis, ()	N/A

Empty parenthesis and VC++

Visual C++ 9 through 12 handles empty parenthesis correctly when it is passed as preprocessor data but Visual C++ 8 (Visual Studio 2005) becomes confused by empty parenthesis passed as possibly parameter data between macros, so this technique for passing empty data as seqs or tuples should be avoided if the compiler being used could be VC++8. A C++ standard comforming preprocessor should have no problems handling data as an empty parenthesis.

Boost PP reentrant versions

A number of macros in VMD have equivalent reentrant versions which are meant to be used in a BOOST_PP_WHILE loop. These are versions which have an underscore D suffix and have the exact same functionality as their unsuffixed equivalents. They can be used in BOOST_PP_WHILE loops to provide slightly quicker preprocessing but, as the documentation for BOOST_PP_WHILE and BOOST_PP_WHILE_##d explain, they do not have to be used.

These macros are:

Identifiers

- BOOST_VMD_IDENTIFIER_D
- BOOST_VMD_BEGIN_IDENTIFIER_D
- BOOST_VMD_AFTER_IDENTIFIER_D
- BOOST_VMD_IS_BEGIN_IDENTIFIER_D
- BOOST_VMD_IS_IDENTIFIER_D
- BOOST_VMD_ASSERT_IS_IDENTIFIER_D

Lists

- BOOST_VMD_LIST_D
- BOOST_VMD_BEGIN_LIST_D
- BOOST_VMD_AFTER_LIST_D
- BOOST_VMD_IS_BEGIN_LIST_D
- BOOST_VMD_IS_LIST_D
- BOOST_VMD_IS_EMPTY_LIST_D
- BOOST_VMD_ASSERT_IS_LIST_D

Seqs

- BOOST_VMD_SEQ_D
- BOOST_VMD_BEGIN_SEQ_D
- BOOST_VMD_AFTER_SEQ_D
- BOOST_VMD_IS_BEGIN_SEQ_D
- BOOST_VMD_IS_SEQ_D
- BOOST_VMD_ASSERT_IS_SEQ_D

Input as dynamic types

Within the constraints based on the top-level types which VMD can parse, the libraries gives the end-user the ability to design macros with dynamic v-types. By this I mean that a macro could be designed to handle different v-types based on some documented agreement of different combinations of macro input meaning slightly different things. Add to this the ability to design such macros with variadic parameters and we have a preprocessor system of macro creation which to a lesser extent rivals the DSELS of template metaprogramming. Of course the preprocessor is not nearly as flexible as C++ templates, but still the sort of preprocessor metaprogramming one could do with VMD, and the underlying Boost PP, in creating flexible macros which can handle different combinations of v-types is very interesting.

Of course macros need to be usable by an end-user so the syntactical ability of v-types and v-type sequences to represent different types of input data must be balanced against ease of use and understanding when using a macro. But because certain v-type sequences can mimic C++ function calls to some extent it is possible to represent macros as a language closer to C++ with VMD.

What is important when designing a macro in which you parse input to decide which type of data the invoker is passing to your macro is that you are aware of the constraints when parsing a v-type. As an example if you design a macro where some input can either be a v-number, a v-identifier, or some none v-type top-level input then attempting to parse the data to see if it is a number or identifier could fail with a preprocessor error and nullify your design if the data is not a v-type. So designing a macro with VMD v-types in mind often means restricting data to parseable top-level v-types.

Visual C++ gotchas in VMD

I have discussed throughout the documentation areas of VMD which need to be considered when using Microsoft's Visual C++ compilers. The VMD library supports VC++ versions 8 through the latest 12. These correspond to Visual Studio 20056 through the current Visual Studio 2013.

I will give here fairly briefly the VC++ quirks which should be taken into account when using VMD. These quirks exist because the VC++ compiler does not have a C++ standard conforming preprocessor. More specifically the VC++ compiler does not follow all of the rules correctly for expanding a macro when a macro is invoked. Here is a list for things to consider when using VMD with VC++:

- The BOOST_VMD_IS_EMPTY macro will expand erroneously to 1 if the input resolves to a function-like macro name, which when it is called with an empty parameter expands to a tuple.
- The BOOST_VMD_ASSERT macro, and the corresponding individual VMD ASSERT macros for the various VMD data types, do not cause an immediate compiler error, but instead generate invalid C++ if the ASSERT occurs.
- When the BOOST_VMD_ASSERT macro, or one of the corresponding individual VMD ASSERT macros for the various VMD data types, does not generate an error, and the macro in which it is being used does generate some output, it is necessary to use BOOST_PP_CAT to concatenate the empty result of the VMD ASSERT macro with the normally generated output to correctly generate the final expansion of the macro in which the VMD ASSERT occurs.
- When using BOOST_VMD_EMPTY following some non-empty expansion, or when using BOOST_VMD_IDENTITY, the value returned needs to be concatenated using BOOST_PP_CAT with an empty value. You can use BOOST_VMD_IDENTITY_RESULT to accomplish this transparently.
- Avoid using an empty parenthesis to pass no data as a tuple or seq if VC++8 might be used as the compiler.

Variadic Macro Data Reference

Header <boost/vmd/array.hpp>

```
BOOST_VMD_ARRAY(param)
BOOST_VMD_BEGIN_ARRAY(param)
BOOST_VMD_AFTER_ARRAY(param)
BOOST_VMD_IS_BEGIN_ARRAY(param)
BOOST_VMD_IS_ARRAY(array)
BOOST_VMD_IS_EMPTY_ARRAY_IRESULT(param)
BOOST_VMD_IS_EMPTY_ARRAY(param)
BOOST_VMD_ASSERT_IS_ARRAY(array)
```

Macro BOOST_VMD_ARRAY

BOOST_VMD_ARRAY — Expands to a tuple of the beginning array, and the preprocessor tokens after the beginning array, of a macro parameter.

Synopsis

```
// In header: <boost/vmd/array.hpp>

BOOST_VMD_ARRAY(param)
```

Description

param = a macro parameter.

returns = the result is a tuple of two elements. If the param does not start with an array, both elements of the tuple are empty. If the param does start with an array, the first element is the array and the second element is the preprocessor tokens after the beginning array.

Macro BOOST_VMD_BEGIN_ARRAY

BOOST_VMD_BEGIN_ARRAY — Expands to the beginning array of a macro parameter.

Synopsis

```
// In header: <boost/vmd/array.hpp>

BOOST_VMD_BEGIN_ARRAY(param)
```

Description

param = a macro parameter.

returns = A beginning array of the macro parameter. If the param does not start with an array, expands to nothing.

Macro BOOST_VMD_AFTER_ARRAY

BOOST_VMD_AFTER_ARRAY — Expands to the preprocessor tokens after the beginning array of a macro parameter.

Synopsis

```
// In header: <boost/vmd/array.hpp>

BOOST_VMD_AFTER_ARRAY(param)
```

Description

param = a macro parameter.

returns = The preprocessor tokens after the beginning array of the macro parameter. If the param does not start with an array, expands to nothing.

Macro BOOST_VMD_IS_BEGIN_ARRAY

BOOST_VMD_IS_BEGIN_ARRAY — Tests whether a parameter begins with an array.

Synopsis

```
// In header: <boost/vmd/array.hpp>

BOOST_VMD_IS_BEGIN_ARRAY(param)
```

Description

param = a macro parameter.

returns = 1 if the param begins with an array, 0 if it does not.

Macro BOOST_VMD_IS_ARRAY

BOOST_VMD_IS_ARRAY — Determines if a parameter is a Boost pplib array.

Synopsis

```
// In header: <boost/vmd/array.hpp>

BOOST_VMD_IS_ARRAY(array)
```

Description

The macro checks that the parameter is a pplib array. It returns 1 if it is an array, else if returns 0.

The macro works through variadic macro support.

array = a possible pplib array.

returns = 1 if it an array, else returns 0.

Macro BOOST_VMD_IS_EMPTY_ARRAY_IRESULT

BOOST_VMD_IS_EMPTY_ARRAY_IRESULT

Synopsis

```
// In header: <boost/vmd/array.hpp>

BOOST_VMD_IS_EMPTY_ARRAY_IRESULT(param)
```

Macro BOOST_VMD_IS_EMPTY_ARRAY

BOOST_VMD_IS_EMPTY_ARRAY — Tests whether an array is an empty Boost PP array.

Synopsis

```
// In header: <boost/vmd/array.hpp>

BOOST_VMD_IS_EMPTY_ARRAY(param)
```

Description

An empty Boost PP array is a two element tuple where the first size element is 0 and the second tuple element is empty.

param = a preprocessor parameter

returns = 1 if the param is an empty Boost PP array 0 if it is not.

Macro BOOST_VMD_ASSERT_IS_ARRAY

BOOST_VMD_ASSERT_IS_ARRAY — Asserts that the parameter is a pplib array.

Synopsis

```
// In header: <boost/vmd/array.hpp>

BOOST_VMD_ASSERT_IS_ARRAY(array)
```

Description

The macro checks that the parameter is a pplib array. If it is not a pplib array, it forces a compiler error.

The macro works through variadic macro support.

The macro normally checks for a pplib array only in debug mode. However an end-user can force the macro to check or not check by defining the macro BOOST_VMD_ASSERT_DATA to 1 or 0 respectively.

array = a possible pplib array.

returns = Normally the macro returns nothing. If the parameter is a pplib array, nothing is output. For VC++, because there is no sure way of forcing a compiler error from within a macro without producing output, if the parameter is not a pplib array the macro forces a compiler error by outputting invalid C++. For all other compilers a compiler error is forced without producing output if the parameter is not a pplib array.

Header <boost/vmd/assert.hpp>

```
BOOST_VMD_ASSERT(...)
```

Macro BOOST_VMD_ASSERT

BOOST_VMD_ASSERT — Conditionally causes an error to be generated.

Synopsis

```
// In header: <boost/vmd/assert.hpp>

BOOST_VMD_ASSERT(...)
```

Description

... = variadic parameters, maximum of 2 will be considered. Any variadic parameters beyond the maximum of 2 are just ignored.

The first variadic parameter is:

cond = A condition that determines whether an assertion occurs. Valid values range from 0 to BOOST_PP_LIMIT_MAG.

The second variadic parameter (optional) is:

errstr = An error string for generating a compiler error when using the VC++ compiler. The VC++ compiler is incapable of producing a preprocessor error so when the 'cond' is 0, a compiler error is generated by outputting C++ code in the form of:

typedef char errstr[-1];

returns = If cond expands to 0, this macro causes an error. Otherwise, it expands to nothing. For all compilers other than Visual C++ the error is a preprocessing error. For Visual C++ the error is caused by output invalid C++: this error could be masked if the invalid output is ignored by a macro which invokes this macro.

Header <boost/vmd/empty.hpp>

```
BOOST_VMD_EMPTY(...)
```

Macro BOOST_VMD_EMPTY

BOOST_VMD_EMPTY

Synopsis

```
// In header: <boost/vmd/empty.hpp>

BOOST_VMD_EMPTY(...)
```

Header <boost/vmd/identifier.hpp>

```
BOOST_VMD_IDENTIFIER(parameter, ...)
BOOST_VMD_IDENTIFIER_D(d, parameter, ...)
BOOST_VMD_BEGIN_IDENTIFIER(parameter, ...)
BOOST_VMD_BEGIN_IDENTIFIER_D(d, parameter, ...)
BOOST_VMD_AFTER_IDENTIFIER(parameter, ...)
BOOST_VMD_AFTER_IDENTIFIER_D(d, parameter, ...)
BOOST_VMD_IS_BEGIN_IDENTIFIER(parameter, ...)
BOOST_VMD_IS_BEGIN_IDENTIFIER_D(d, parameter, ...)
```

Macro BOOST_VMD_IDENTIFIER

BOOST_VMD_IDENTIFIER — Expands to the index of a beginning identifier found and the preprocessor tokens after the beginning identifier.

Synopsis

```
// In header: <boost/vmd/identifier.hpp>

BOOST_VMD_IDENTIFIER(parameter, ...)
```

Description

parameter = a macro parameter. ... = variadic parameters, maximum of 3

The first variadic parameter is:

key(s) = The data may take one of two forms: it is either a single C++ identifier as a unique 'key' or a Boost PP tuple of the unique 'key's.

The idea of a unique 'key' is not to duplicate any key that another library may use.

A unique key should not begin with an underscore. It may end with an underscore only if the value to be tested does not begin with an underscore. It may contain underscores.

Two different suggested ways to generate a unique key are: 1) Use a specific mnemonic for a particular module and concatenate an increasing numeric value, perhaps separated by an underscore, to it each time you use it, ie. TTI_0, TTI_1, TTI_2 etc. 2) use a GUID (128 bit unique number) generated by the OS or software, prefixed by some alphabetic.

The 'identifier' to be tested against is specified by having the end-user defining an object-like macro, which does not expand to anything, whose form is:

#define BOOST_VMD_MAP_'key''identifier'

where 'key' is the key, or one of the keys, passed in this parameter, 'identifier' is a value to be tested against.

The second variadic parameter (optional) is: 1) a Boost PP sequence where each element in the sequence contains the key(s) of subsequent identifiers, where a 'key' is explained in the first variadic parameter. In other words each sequence element contains either a single key or a tuple of keys, ie '(key)' or '((key,key,key))'. The maximum number of sequence elements allowed is 4. OR 2) a number represented a maximum amount of subsequent Boost PP numbers following the identifier. The maximum number for this amount is 5. If the second optional parameter is not specified a beginning identifier will also always be found if followed by a single Boost PP number, a Boost PP tuple, or the end of the parameter. If the second optional parameter is a Boost PP sequence a beginning identifier will also be found if, subsequent to matching 0 or more further keys, a Boost PP tuple or the end of the parameter is found.

If the second optional parameter is a number a beginning identifier will also be found if, subsequent to matching 0 or more Boost PP numbers, a Boost PP tuple or the end of the parameter is found.

The third variadic parameter (optional) is a number represented a maximum amount of Boost PP numbers following the subsequent identifiers specified by the second variadic parameter. The maximum number for this amount is 5.

If the second and third variadic parameters are specified, a beginning identifier will also be found if, subsequent to matching further keys followed by further numbers, a Boost PP tuple or the end of the parameter is found.

returns = the result is a tuple of two elements. If a beginning identifier is not found, the first element is 0 and the second element of the tuple is empty. If a beginning identifier is found, the first element is the index, starting with 1, of the identifier and the second element is the preprocessor tokens after the identifier.

Macro BOOST_VMD_IDENTIFIER_D

BOOST_VMD_IDENTIFIER_D

Synopsis

```
// In header: <boost/vmd/identifier.hpp>

BOOST_VMD_IDENTIFIER_D(d, parameter, ...)
```

Macro BOOST_VMD_BEGIN_IDENTIFIER

BOOST_VMD_BEGIN_IDENTIFIER — Expands to the index of a beginning identifier of a macro parameter.

Synopsis

```
// In header: <boost/vmd/identifier.hpp>

BOOST_VMD_BEGIN_IDENTIFIER(parameter, ...)
```

Description

parameter = a macro parameter. ... = variadic parameters, maximum of 3

The first variadic parameter is:

key(s) = The data may take one of two forms: it is either a single C++ identifier as a unique 'key' or a Boost PP tuple of the unique 'key's.

The idea of a unique 'key' is not to duplicate any key that another library may use.

A unique key should not begin with an underscore. It may end with an underscore only if the value to be tested does not begin with an underscore. It may contain underscores.

Two different suggested ways to generate a unique key are: 1) Use a specific mnemonic for a particular module and concatenate an increasing numeric value, perhaps separated by an underscore, to it each time you use it, ie. TTI_0, TTI_1, TTI_2 etc. 2) use a GUID (128 bit unique number) generated by the OS or software, prefixed by some alphabetic.

The 'identifier' to be tested against is specified by having the end-user defining an object-like macro, which does not expand to anything, whose form is:

#define BOOST_VMD_MAP_'key''identifier'

where 'key' is the key, or one of the keys, passed in this parameter, 'identifier' is a value to be tested against.

The second variadic parameter (optional) is: 1) a Boost PP sequence where each element in the sequence contains the key(s) of subsequent identifiers, where a 'key' is explained in the first variadic parameter. In other words each sequence element contains either a single key or a tuple of keys, ie '(key)' or '((key,key,key))'. The maximum number of sequence elements allowed is 4. OR 2) a number represented a maximum amount of subsequent Boost PP numbers following the identifier. The maximum number for this amount is 5. If the second optional parameter is not specified a beginning identifier will also always be found if followed by a single Boost PP number, a Boost PP tuple, or the end of the parameter. If the second optional parameter is a Boost PP sequence a beginning identifier will also be found if, subsequent to matching 0 or more further keys, a Boost PP tuple or the end of the parameter is found. If the second optional parameter is a number a beginning identifier will also be found if, subsequent to matching 0 or more Boost PP numbers, a Boost PP tuple or the end of the parameter is found.

The third variadic parameter (optional) is a number represented a maximum amount of Boost PP numbers following the subsequent identifiers specified by the second variadic parameter. The maximum number for this amount is 5.

If the second and third variadic parameters are specified, a beginning identifier will also be found if, subsequent to matching further keys followed by further numbers, a Boost PP tuple or the end of the parameter is found.

returns = expands to the index, starting with 1, of the particular identifier it matches, otherwise expands to 0.

Macro BOOST_VMD_BEGIN_IDENTIFIER_D

BOOST_VMD_BEGIN_IDENTIFIER_D

Synopsis

```
// In header: <boost/vmd/identifier.hpp>

BOOST_VMD_BEGIN_IDENTIFIER_D(d, parameter, ...)
```

Macro BOOST_VMD_AFTER_IDENTIFIER

BOOST_VMD_AFTER_IDENTIFIER — Expands to the preprocessor tokens after the identifier of a macro parameter.

Synopsis

```
// In header: <boost/vmd/identifier.hpp>

BOOST_VMD_AFTER_IDENTIFIER(parameter, ...)
```

Description

parameter = a macro parameter. ... = variadic parameters, maximum of 3

The first variadic parameter is:

key(s) = The data may take one of two forms: it is either a single C++ identifier as a unique 'key' or a Boost PP tuple of the unique 'key's.

The idea of a unique 'key' is not to duplicate any key that another library may use.

A unique key should not begin with an underscore. It may end with an underscore only if the value to be tested does not begin with an underscore. It may contain underscores.

Two different suggested ways to generate a unique key are: 1) Use a specific mnemonic for a particular module and concatenate an increasing numeric value, perhaps separated by an underscore, to it each time you use it, ie. TTI_0, TTI_1, TTI_2 etc. 2) use a GUID (128 bit unique number) generated by the OS or software, prefixed by some alphabetic.

The 'identifier' to be tested against is specified by having the end-user defining an object-like macro, which does not expand to anything, whose form is:

#define BOOST_VMD_MAP_'key''identifier'

where 'key' is the key, or one of the keys, passed in this parameter, 'identifier' is a value to be tested against.

The second variadic parameter (optional) is: 1) a Boost PP sequence where each element in the sequence contains the key(s) of subsequent identifiers, where a 'key' is explained in the first variadic parameter. In other words each sequence element contains either a single key or a tuple of keys, ie '(key)' or '((key,key,key))'. The maximum number of sequence elements allowed is 4. OR 2) a number represented a maximum amount of subsequent Boost PP numbers following the identifier. The maximum number for this amount is 5. If the second optional parameter is not specified a beginning identifier will also always be found if followed by a single Boost PP number, a Boost PP tuple, or the end of the parameter. If the second optional parameter is a Boost PP sequence a beginning identifier will also be found if, subsequent to matching 0 or more further keys, a Boost PP tuple or the end of the parameter is found. If the second optional parameter is a number a beginning identifier will also be found if, subsequent to matching 0 or more Boost PP numbers, a Boost PP tuple or the end of the parameter is found.

The third variadic parameter (optional) is a number represented a maximum amount of Boost PP numbers following the subsequent identifiers specified by the second variadic parameter. The maximum number for this amount is 5.

If the second and third variadic parameters are specified, a beginning identifier will also be found if, subsequent to matching further keys followed by further numbers, a Boost PP tuple or the end of the parameter is found.

returns = expands to the preprocessor tokens after the identifier. If the identifier is not found, expands to nothing.

Macro BOOST_VMD_AFTER_IDENTIFIER_D

BOOST_VMD_AFTER_IDENTIFIER_D

Synopsis

```
// In header: <boost/vmd/identifier.hpp>

BOOST_VMD_AFTER_IDENTIFIER_D(d, parameter, ...)
```

Macro BOOST_VMD_IS_BEGIN_IDENTIFIER

BOOST_VMD_IS_BEGIN_IDENTIFIER — Tests whether a parameter begins with an identifier.

Synopsis

```
// In header: <boost/vmd/identifier.hpp>

BOOST_VMD_IS_BEGIN_IDENTIFIER(parameter, ...)
```

Description

parameter = a macro parameter. ... = variadic parameters, maximum of 3

The first variadic parameter is:

key(s) = The data may take one of two forms: it is either a single C++ identifier as a unique 'key' or a Boost PP tuple of the unique 'key's.

The idea of a unique 'key' is not to duplicate any key that another library may use.

A unique key should not begin with an underscore. It may end with an underscore only if the value to be tested does not begin with an underscore. It may contain underscores.

Two different suggested ways to generate a unique key are: 1) Use a specific mnemonic for a particular module and concatenate an increasing numeric value, perhaps separated by an underscore, to it each time you use it, ie. TTI_0, TTI_1, TTI_2 etc. 2) use a GUID (128 bit unique number) generated by the OS or software, prefixed by some alphabetic.

The 'identifier' to be tested against is specified by having the end-user defining an object-like macro, which does not expand to anything, whose form is:

#define BOOST_VMD_MAP_'key''identifier'

where 'key' is the key, or one of the keys, passed in this parameter, 'identifier' is a value to be tested against.

The second variadic parameter (optional) is: 1) a Boost PP sequence where each element in the sequence contains the key(s) of subsequent identifiers, where a 'key' is explained in the first variadic parameter. In other words each sequence element contains either a single key or a tuple of keys, ie '(key)' or '((key,key,key))'. The maximum number of sequence elements allowed is 4. OR 2) a number represented a maximum amount of subsequent Boost PP numbers following the identifier. The maximum number for this amount is 5. If the second optional parameter is not specified a beginning identifier will also always be found if followed by a single Boost PP number, a Boost PP tuple, or the end of the parameter. If the second optional parameter is a Boost PP sequence a beginning identifier will also be found if, subsequent to matching 0 or more further keys, a Boost PP tuple or the end of the parameter is found. If the second optional parameter is a number a beginning identifier will also be found if, subsequent to matching 0 or more Boost PP numbers, a Boost PP tuple or the end of the parameter is found.

The third variadic parameter (optional) is a number represented a maximum amount of Boost PP numbers following the subsequent identifiers specified by the second variadic parameter. The maximum number for this amount is 5.

If the second and third variadic parameters are specified, a beginning identifier will also be found if, subsequent to matching further keys followed by further numbers, a Boost PP tuple or the end of the parameter is found.

returns = 1 if the param begins with an identifier, 0 if it does not.

Macro BOOST_VMD_IS_BEGIN_IDENTIFIER_D

BOOST_VMD_IS_BEGIN_IDENTIFIER_D

Synopsis

```
// In header: <boost/vmd/identifier.hpp>

BOOST_VMD_IS_BEGIN_IDENTIFIER_D(d, parameter, ...)
```

Header <boost/vmd/identity.hpp>

```


BOOST_VMD_IDENTITY(item)
BOOST_VMD_IDENTITY_RESULT(item)
```

Macro BOOST_VMD_IDENTITY

BOOST_VMD_IDENTITY

Synopsis

```
// In header: <boost/vmd/identity.hpp>

BOOST_VMD_IDENTITY(item)
```

Macro BOOST_VMD_IDENTITY_RESULT

BOOST_VMD_IDENTITY_RESULT

Synopsis

```
// In header: <boost/vmd/identity.hpp>

BOOST_VMD_IDENTITY_RESULT(item)
```

Header <boost/vmd/is_begin_tuple.hpp>

```


BOOST_VMD_IS_BEGIN_TUPLE(...)
```

Macro BOOST_VMD_IS_BEGIN_TUPLE

BOOST_VMD_IS_BEGIN_TUPLE — Tests whether a parameter begins with a tuple.

Synopsis

```
// In header: <boost/vmd/is_begin_tuple.hpp>

BOOST_VMD_IS_BEGIN_TUPLE(...)
```

Description

The macro checks to see if the parameter begins with a tuple.

.... = variadic param(s)

returns = 1 if the param begins with a tuple, 0 if it does not.

The macro works through variadic macro support.

The code is taken from a posting by Paul Mensonides.

This macro is not a test for only a tuple since the parameter may have other tokens following the tuple and it will still return 1.

Header <boost/vmd/is_empty.hpp>

```
BOOST_VMD_IS_EMPTY(...)
BOOST_VMD_ASSERT_IS_EMPTY(...)
```

Macro BOOST_VMD_IS_EMPTY

BOOST_VMD_IS_EMPTY — Tests whether its input is empty or not.

Synopsis

```
// In header: <boost/vmd/is_empty.hpp>

BOOST_VMD_IS_EMPTY(...)
```

Description

The macro checks to see if the input is empty or not. It returns 1 if the input is empty, else returns 0.

The macro is a variadic macro taking any input and works through variadic macro support.

The macro is not perfect, and can not be so. The problem area is if the input to be checked is a function-like macro name, in which case either a compiler error can result or a false result can occur.

This macro is a replacement, using variadic macro support, for the undocumented macro BOOST_PP_IS_EMPTY in the Boost pplib. The code is taken from a posting by Paul Mensonides of a variadic version for BOOST_PP_IS_EMPTY, and changed in order to also support VC++.

.... = variadic input

returns = 1 if the input is empty, 0 if it is not

It is recommended to append BOOST_PP_EMPTY() to whatever input is being tested in order to avoid possible warning messages from some compilers about no parameters being passed to the macro when the input is truly empty.

Macro BOOST_VMD_ASSERT_IS_EMPTY

BOOST_VMD_ASSERT_IS_EMPTY

Synopsis

```
// In header: <boost/vmd/is_empty.hpp>

BOOST_VMD_ASSERT_IS_EMPTY(...)
```

Header <boost/vmd/is_identifier.hpp>

```
BOOST_VMD_IS_IDENTIFIER(parameter, keys)
BOOST_VMD_IS_IDENTIFIER_D(d, parameter, keys)
BOOST_VMD_ASSERT_IS_IDENTIFIER(parameter, keys)
BOOST_VMD_ASSERT_IS_IDENTIFIER_D(d, parameter, keys)
```

Macro BOOST_VMD_IS_IDENTIFIER

BOOST_VMD_IS_IDENTIFIER — Tests whether a parameter is the same as a particular identifier.

Synopsis

```
// In header: <boost/vmd/is_identifier.hpp>

BOOST_VMD_IS_IDENTIFIER(parameter, keys)
```

Description

The macro checks to see if the 'parameter' is equal to a particular identifier. meaning that it is the exact same preprocessing token. An identifier which is checked is mapped by a key (see below), which must be unique each time this macro is invoked.

It returns the index, starting with 1, of the identifier which it matches, else returns 0. The parameter must be a C++ identifier, ie. it must consist of letters, numbers, and underscores.

parameter = a C++ identifier to test keys = The variadic data may take one of two forms: it is either a C++ identifier as a unique 'key' (see below) or a Boost PP tuple of the unique 'key's.

The idea of a unique 'key' is not to duplicate any key that another library may use.

A unique key should not begin with an underscore. It may end with an underscore only if the value to be tested does not begin with an underscore. It may contain underscores.

Two different suggested ways to generate a unique key are: 1) Use a specific mnemonic for a particular module and concatenate an increasing numeric value, perhaps separated by an underscore, to it each time you use it, ie. TTI_0, TTI_1, TTI_2 etc. 2) use a GUID (128 bit unique number) generated by the OS or software, prefixed by some alphabetic.

The 'identifier' to be tested against is specified by having the end-user defining an object-like macro, which does not expand to anything, whose form is:

#define BOOST_VMD_MAP_'key''identifier'

where 'key' is the key, or one of the keys, passed in this parameter, 'identifier' is a value to be tested against.

returns = expands to the index, starting with 1, of the particular identifier it matches, otherwise expands to 0.

As an example, let us suppose that within a library called 'plane_geometry' I want to use the BOOST_VMD_IS_IDENTIFIER macro 4 times, checking whether a parameter is equal to 'SQUARE' the first time, 'TRIANGLE' the second time, 'CIRCLE' the third time, and any one of them the fourth time.

First I write my object-like macros

#define BOOST_VMD_MAP_PLANEGEOMETRY_1_SQUARE #define BOOST_VMD_MAP_PLANEGEOMETRY_2_TRIANGLE #define BOOST_VMD_MAP_PLANEGEOMETRY_3_CIRCLE

If I want to check if some preprocessor argument 'x' is SQUARE I invoke:

BOOST_VMD_IS_IDENTIFIER(x,PLANEGEOMETRY_1_) or BOOST_VMD_IS_IDENTIFIER(x,(PLANEGEOMETRY_1_))

If I want to check if some preprocessor argument 'x' is TRIANGLE I invoke:

BOOST_VMD_IS_IDENTIFIER(x,PLANEGEOMETRY_2_) or BOOST_VMD_IS_IDENTIFIER(x,(PLANEGEOMETRY_2_))

If I want to check if some preprocessor argument 'x' is CIRCLE I invoke:

BOOST_VMD_IS_IDENTIFIER(x,PLANEGEOMETRY_3_) or BOOST_VMD_IS_IDENTIFIER(x,(PLANEGEOMETRY_3_))

If I want to check if some preprocessor argument 'x' is either a CIRCLE, TRIANGLE, or SQUARE I invoke:

BOOST_VMD_IS_IDENTIFIER(x,(PLANEGEOMETRY_1_,PLANEGEOMETRY_2_,PLANEGEOMETRY_3_))

Macro BOOST_VMD_IS_IDENTIFIER_D

BOOST_VMD_IS_IDENTIFIER_D

Synopsis

```
// In header: <boost/vmd/is_identifier.hpp>

BOOST_VMD_IS_IDENTIFIER_D(d, parameter, keys)
```

Macro BOOST_VMD_ASSERT_IS_IDENTIFIER

BOOST_VMD_ASSERT_IS_IDENTIFIER

Synopsis

```
// In header: <boost/vmd/is_identifier.hpp>

BOOST_VMD_ASSERT_IS_IDENTIFIER(parameter, keys)
```

Macro BOOST_VMD_ASSERT_IS_IDENTIFIER_D

BOOST_VMD_ASSERT_IS_IDENTIFIER_D

Synopsis

```
// In header: <boost/vmd/is_identifier.hpp>

BOOST_VMD_ASSERT_IS_IDENTIFIER_D(d, parameter, keys)
```

Header <boost/vmd/is_number.hpp>

```


BOOST_VMD_IS_NUMBER(ppident)
BOOST_VMD_ASSERT_IS_NUMBER(ppident)
```

Macro BOOST_VMD_IS_NUMBER

BOOST_VMD_IS_NUMBER — Tests whether a parameter is a Boost PP number.

Synopsis

```
// In header: <boost/vmd/is_number.hpp>

BOOST_VMD_IS_NUMBER(ppident)
```

Description

The macro checks to see if a parameter is a Boost PP number. A Boost PP number is a value from 0 to 256.

ppident = a preprocessor identifier

returns = 1 if the param is a Boost PP number, 0 if it is not.

The macro works through variadic macro support. The ppident can be either:

1) A preprocessor identifier, alphanumeric or underscore characters. 2) An empty value, returns 0. 3) A set of beginning parens, returns 0.

If it is not one of these possibilities a compiler error will occur.

Macro BOOST_VMD_ASSERT_IS_NUMBER

BOOST_VMD_ASSERT_IS_NUMBER

Synopsis

```
// In header: <boost/vmd/is_number.hpp>

BOOST_VMD_ASSERT_IS_NUMBER(ppident)
```

Header <boost/vmd/is_parens_empty.hpp>

```


BOOST_VMD_IS_PARENS_EMPTY(param)
```

Macro BOOST_VMD_IS_PARENS_EMPTY

BOOST_VMD_IS_PARENS_EMPTY — Determines if the input is a set of parens with no data.

Synopsis

```
// In header: <boost/vmd/is_parens_empty.hpp>

BOOST_VMD_IS_PARENS_EMPTY(param)
```

Description

param = a macro parameter.

returns = 1 if the input is a set of parens with no data, else returns 0.

A set of parens with no data may be:

1) a tuple whose size is a single element which is empty or

2) a single element seq whose data is empty

Header <boost/vmd/list.hpp>

```
BOOST_VMD_LIST(param)
BOOST_VMD_LIST_D(d, param)
BOOST_VMD_BEGIN_LIST(param)
BOOST_VMD_BEGIN_LIST_D(d, param)
BOOST_VMD_AFTER_LIST(param)
BOOST_VMD_AFTER_LIST_D(d, param)
BOOST_VMD_IS_BEGIN_LIST(param)
BOOST_VMD_IS_BEGIN_LIST_D(d, param)
BOOST_VMD_IS_LIST(param)
BOOST_VMD_IS_LIST_D(d, param)
BOOST_VMD_IS_EMPTY_LIST_IRESULT(param)
BOOST_VMD_IS_EMPTY_LIST_D_IRESULT(d, param)
BOOST_VMD_IS_EMPTY_LIST(param)
BOOST_VMD_IS_EMPTY_LIST_D(d, param)
BOOST_VMD_ASSERT_IS_LIST(param)
BOOST_VMD_ASSERT_IS_LIST_D(d, param)
```

Macro BOOST_VMD_LIST

BOOST_VMD_LIST — Expands to a tuple of the beginning list, and the preprocessor tokens after the beginning list, of a macro parameter.

Synopsis

```
// In header: <boost/vmd/list.hpp>

BOOST_VMD_LIST(param)
```

Description

param = input possibly beginning with a Boost PP list.

returns = the result is a tuple of two elements. If the param does not start with a list, both elements of the tuple are empty. If the param does start with a list, the first element is the list and the second element is the preprocessor tokens after the beginning list.

Macro BOOST_VMD_LIST_D

BOOST_VMD_LIST_D

Synopsis

```
// In header: <boost/vmd/list.hpp>

BOOST_VMD_LIST_D(d, param)
```

Macro BOOST_VMD_BEGIN_LIST

BOOST_VMD_BEGIN_LIST — Expands to the beginning list of a macro parameter.

Synopsis

```
// In header: <boost/vmd/list.hpp>

BOOST_VMD_BEGIN_LIST(param)
```

Description

param = input possibly beginning with a Boost PP list.

returns = A beginning list of the macro parameter. If the param does not start with a list, expands to nothing.

Macro BOOST_VMD_BEGIN_LIST_D

BOOST_VMD_BEGIN_LIST_D

Synopsis

```
// In header: <boost/vmd/list.hpp>

BOOST_VMD_BEGIN_LIST_D(d, param)
```

Macro BOOST_VMD_AFTER_LIST

BOOST_VMD_AFTER_LIST — Expands to the preprocessor tokens after the beginning list of a macro parameter.

Synopsis

```
// In header: <boost/vmd/list.hpp>

BOOST_VMD_AFTER_LIST(param)
```

Description

param = input possibly beginning with a Boost PP list.

returns = The preprocessor tokens after the beginning list of the macro parameter. If the param does not start with a list, expands to nothing.

Macro BOOST_VMD_AFTER_LIST_D

BOOST_VMD_AFTER_LIST_D

Synopsis

```
// In header: <boost/vmd/list.hpp>

BOOST_VMD_AFTER_LIST_D(d, param)
```

Macro BOOST_VMD_IS_BEGIN_LIST

BOOST_VMD_IS_BEGIN_LIST — Tests whether a parameter begins with a list.

Synopsis

```
// In header: <boost/vmd/list.hpp>

BOOST_VMD_IS_BEGIN_LIST(param)
```

Description

param = input possibly beginning with a Boost PP list.

returns = 1 if the param begins with a list, 0 if it does not.

Macro BOOST_VMD_IS_BEGIN_LIST_D

BOOST_VMD_IS_BEGIN_LIST_D

Synopsis

```
// In header: <boost/vmd/list.hpp>

BOOST_VMD_IS_BEGIN_LIST_D(d, param)
```

Macro BOOST_VMD_IS_LIST

BOOST_VMD_IS_LIST — Determines if a parameter is a Boost pplib list.

Synopsis

```
// In header: <boost/vmd/list.hpp>

BOOST_VMD_IS_LIST(param)
```

Description

The macro checks that the parameter is a pplib list.

The macro works through variadic macro support. It returns 1 if it is a list, else if returns 0.

param = input as a possible Boost PP list.

returns = 1 if it a list, else returns 0.

Macro BOOST_VMD_IS_LIST_D

BOOST_VMD_IS_LIST_D

Synopsis

```
// In header: <boost/vmd/list.hpp>

BOOST_VMD_IS_LIST_D(d, param)
```

Macro BOOST_VMD_IS_EMPTY_LIST_IRESULT

BOOST_VMD_IS_EMPTY_LIST_IRESULT

Synopsis

```
// In header: <boost/vmd/list.hpp>

BOOST_VMD_IS_EMPTY_LIST_IRESULT(param)
```

Macro BOOST_VMD_IS_EMPTY_LIST_D_IRESULT

BOOST_VMD_IS_EMPTY_LIST_D_IRESULT

Synopsis

```
// In header: <boost/vmd/list.hpp>

BOOST_VMD_IS_EMPTY_LIST_D_IRESULT(d, param)
```

Macro BOOST_VMD_IS_EMPTY_LIST

BOOST_VMD_IS_EMPTY_LIST — Tests whether a list is an empty Boost PP list.

Synopsis

```
// In header: <boost/vmd/list.hpp>

BOOST_VMD_IS_EMPTY_LIST(param)
```

Description

An empty Boost PP list consists of the single identifier 'BOOST_PP_NIL'. This identifier also serves as a list terminator for a non-empty list.

param = a preprocessor parameter

returns = 1 if the param is an empty Boost PP list 0 if it is not.

Macro BOOST_VMD_IS_EMPTY_LIST_D

BOOST_VMD_IS_EMPTY_LIST_D

Synopsis

```
// In header: <boost/vmd/list.hpp>

BOOST_VMD_IS_EMPTY_LIST_D(d, param)
```

Macro BOOST_VMD_ASSERT_IS_LIST

BOOST_VMD_ASSERT_IS_LIST — Asserts that the parameter is a Boost pplib list.

Synopsis

```
// In header: <boost/vmd/list.hpp>

BOOST_VMD_ASSERT_IS_LIST(param)
```

Description

The macro checks that the parameter is a pplib list. If it is not a pplib list, it forces a compiler error.

The macro works through variadic macro support.

The macro normally checks for a pplib list only in debug mode. However an end-user can force the macro to check or not check by defining the macro BOOST_VMD_ASSERT_DATA to 1 or 0 respectively.

param = a possible pplib list.

returns = Normally the macro returns nothing. If the parameter is a pplib list, nothing is output. For VC++, because there is no sure way of forcing a compiler error from within a macro without producing output, if the parameter is not a pplib list the macro forces a compiler error by outputting invalid C++. For all other compilers a compiler error is forced without producing output if the parameter is not a pplib list.

Macro BOOST_VMD_ASSERT_IS_LIST_D

BOOST_VMD_ASSERT_IS_LIST_D

Synopsis

```
// In header: <boost/vmd/list.hpp>

BOOST_VMD_ASSERT_IS_LIST_D(d, param)
```

Header <boost/vmd/number.hpp>

```
BOOST_VMD_NUMBER(...)
BOOST_VMD_BEGIN_NUMBER(...)
BOOST_VMD_AFTER_NUMBER(...)
BOOST_VMD_IS_BEGIN_NUMBER(...)
```

Macro BOOST_VMD_NUMBER

BOOST_VMD_NUMBER — Expands to a tuple of the beginning number and the preprocessor tokens after the beginning number in a parameter.

Synopsis

```
// In header: <boost/vmd/number.hpp>

BOOST_VMD_NUMBER(...)
```

Description

... = One to three variadic parameters. These parameters are:

first = required, the macro parameter to test for a beginning number. second = optional, cnumber an optional digit from 1-5 indicating the maximum amount of consecutive numbers in the parameter. Specifying 1 is not necessary but allowed, as 1 is the default. The actual cnumber may be more than the amount of consecutive numbers which exist. The consecutive numbers must either end the parameter or have a set of parenthesis after them for the beginning number to be found.

OR

isequence an optional Boost PP sequence of 1-5 elements. The elements of the sequence are keys for possibly subsequent identifiers. The keys can be specified as either a single key or as a tuple of keys. The actual number of sequence elements may be more than the amount of consecutive identifiers which exist after the beginning number. The identifiers matching the keys must either end the parameter or have a set of parenthesis after them for the beginning number to be found.

third = optional, isequence if the second optional variadic parameter is a 'cnumber', the third parameter can be an 'isequence'. This allows the beginning number to be followed by one or more other numbers, and then followed by one or more identifiers.

returns = the result is a tuple of two elements. If a beginning number is not found, both elements of the tuple are empty. If a beginning number is found, the first element is the number and the second element is the preprocessor tokens after the number.

The number if found must be in the range of numbers for the Boost preprocessor library, which is 0 to 256.

Macro BOOST_VMD_BEGIN_NUMBER

BOOST_VMD_BEGIN_NUMBER — Expands to the beginning number of a macro parameter.

Synopsis

```
// In header: <boost/vmd/number.hpp>

BOOST_VMD_BEGIN_NUMBER(...)
```

Description

... = One to three variadic parameters. These parameters are:

first = required, the macro parameter to test for a beginning number. second = optional, cnumber an optional digit from 1-5 indicating the maximum amount of consecutive numbers in the parameter. Specifying 1 is not necessary but allowed, as 1 is the default. The actual cnumber may be more than the amount of consecutive numbers which exist. The consecutive numbers must either end the parameter or have a set of parenthesis after them for the beginning number to be found.

OR

isequence an optional Boost PP sequence of 1-5 elements. The elements of the sequence are keys for possibly subsequent identifiers. The keys can be specified as either a single key or as a tuple of keys. The actual number of sequence elements may be more than the amount of consecutive identifiers which exist after the beginning number. The identifiers matching the keys must either end the parameter or have a set of parenthesis after them for the beginning number to be found.

third = optional, isequence if the second optional variadic parameter is a 'cnumber', the third parameter can be an 'isequence'. This allows the beginning number to be followed by one or more other numbers, and then followed by one or more identifiers.

returns = the beginning number of the parameter. If the parameter does not start with a number, the return value is empty.

The number if found must be in the range of numbers for the Boost preprocessor library, which is 0 to 256.

Macro BOOST_VMD_AFTER_NUMBER

BOOST_VMD_AFTER_NUMBER — Expands to the preprocessor tokens after a number of a macro parameter.

Synopsis

```
// In header: <boost/vmd/number.hpp>

BOOST_VMD_AFTER_NUMBER(...)
```

Description

... = One to three variadic parameters. These parameters are:

first = required, the macro parameter to test for a beginning number. second = optional, cnumber an optional digit from 1-5 indicating the maximum amount of consecutive numbers in the parameter. Specifying 1 is not necessary but allowed, as 1 is the default. The actual cnumber may be more than the amount of consecutive numbers which exist. The consecutive numbers must either end the parameter or have a set of parenthesis after them for the beginning number to be found.

OR

isequence an optional Boost PP sequence of 1-5 elements. The elements of the sequence are keys for possibly subsequent identifiers. The keys can be specified as either a single key or as a tuple of keys. The actual number of sequence elements may be more than the amount of consecutive identifiers which exist after the beginning number. The identifiers matching the keys must either end the parameter or have a set of parenthesis after them for the beginning number to be found.

third = optional, isequence if the second optional variadic parameter is a 'cnumber', the third parameter can be an 'isequence'. This allows the beginning number to be followed by one or more other numbers, and then followed by one or more identifiers.

returns = expands to the preprocessor tokens after a number. If the number is not found, expands to nothing.

Macro BOOST_VMD_IS_BEGIN_NUMBER

BOOST_VMD_IS_BEGIN_NUMBER — Tests whether a parameter begins with a number.

Synopsis

```
// In header: <boost/vmd/number.hpp>

BOOST_VMD_IS_BEGIN_NUMBER(...)
```

Description

... = One to three variadic parameters. These parameters are:

first = required, the macro parameter to test for a beginning number. second = optional, cnumber an optional digit from 1-5 indicating the maximum amount of consecutive numbers in the parameter. Specifying 1 is not necessary but allowed, as 1 is the default. The actual cnumber may be more than the amount of consecutive numbers which exist. The consecutive numbers must either end the parameter or have a set of parenthesis after them for the beginning number to be found.

OR

isequence an optional Boost PP sequence of 1-5 elements. The elements of the sequence are keys for possibly subsequent identifiers. The keys can be specified as either a single key or as a tuple of keys. The actual number of sequence elements may be more than the amount of consecutive identifiers which exist after the beginning number. The identifiers matching the keys must either end the parameter or have a set of parenthesis after them for the beginning number to be found.

third = optional, isequence if the second optional variadic parameter is a 'cnumber', the third parameter can be an 'isequence'. This allows the beginning number to be followed by one or more other numbers, and then followed by one or more identifiers.

returns = 1 if the param begins with a number, 0 if it does not.

The number if found must be in the range of numbers for the Boost preprocessor library, which is 0 to 256.

Header <boost/vmd/seq.hpp>

```
BOOST_VMD_SEQ(seq)
BOOST_VMD_SEQ_D(d, seq)
BOOST_VMD_BEGIN_SEQ(param)
BOOST_VMD_BEGIN_SEQ_D(d, param)
BOOST_VMD_AFTER_SEQ(param)
BOOST_VMD_AFTER_SEQ_D(d, param)
BOOST_VMD_IS_BEGIN_SEQ(param)
BOOST_VMD_IS_BEGIN_SEQ_D(d, param)
BOOST_VMD_IS_SEQ(seq)
BOOST_VMD_IS_SEQ_D(d, seq)
BOOST_VMD_ASSERT_IS_SEQ(seq)
BOOST_VMD_ASSERT_IS_SEQ_D(d, seq)
```

Macro BOOST_VMD_SEQ

BOOST_VMD_SEQ — Expands to a tuple of the beginning sequence, and the preprocessor tokens after the beginning sequence, of a macro parameter.

Synopsis

```
// In header: <boost/vmd/seq.hpp>

BOOST_VMD_SEQ(seq)
```

Description

param = a macro parameter.

returns = the result is a tuple of two elements. If the param does not start with a sequence, both elements of the tuple are empty. If the param does start with a sequence, the first element is the beginning sequence and the second element is the preprocessor tokens after the beginning sequence.

Macro BOOST_VMD_SEQ_D

BOOST_VMD_SEQ_D

Synopsis

```
// In header: <boost/vmd/seq.hpp>

BOOST_VMD_SEQ_D(d, seq)
```

Macro BOOST_VMD_BEGIN_SEQ

BOOST_VMD_BEGIN_SEQ — Expands to the beginning sequence of a macro parameter.

Synopsis

```
// In header: <boost/vmd/seq.hpp>

BOOST_VMD_BEGIN_SEQ(param)
```

Description

param = a macro parameter.

returns = A beginning sequence of the macro parameter. If the param does not start with a sequence, expands to nothing.

Macro BOOST_VMD_BEGIN_SEQ_D

BOOST_VMD_BEGIN_SEQ_D

Synopsis

```
// In header: <boost/vmd/seq.hpp>

BOOST_VMD_BEGIN_SEQ_D(d, param)
```

Macro BOOST_VMD_AFTER_SEQ

BOOST_VMD_AFTER_SEQ — Expands to the preprocessor tokens after the beginning sequence of a macro parameter.

Synopsis

```
// In header: <boost/vmd/seq.hpp>

BOOST_VMD_AFTER_SEQ(param)
```

Description

param = a macro parameter.

returns = The preprocessor tokens after the beginning sequence of the macro parameter. If the param does not start with a sequence, expands to nothing.

Macro BOOST_VMD_AFTER_SEQ_D

BOOST_VMD_AFTER_SEQ_D

Synopsis

```
// In header: <boost/vmd/seq.hpp>

BOOST_VMD_AFTER_SEQ_D(d, param)
```

Macro BOOST_VMD_IS_BEGIN_SEQ

BOOST_VMD_IS_BEGIN_SEQ — Tests whether a parameter begins with a sequence.

Synopsis

```
// In header: <boost/vmd/seq.hpp>

BOOST_VMD_IS_BEGIN_SEQ(param)
```

Description

param = a macro parameter.

returns = 1 if the param begins with a sequence, 0 if it does not.

Macro BOOST_VMD_IS_BEGIN_SEQ_D

BOOST_VMD_IS_BEGIN_SEQ_D

Synopsis

```
// In header: <boost/vmd/seq.hpp>

BOOST_VMD_IS_BEGIN_SEQ_D(d, param)
```

Macro BOOST_VMD_IS_SEQ

BOOST_VMD_IS_SEQ — Determines if a parameter is a Boost pplib seq.

Synopsis

```
// In header: <boost/vmd/seq.hpp>

BOOST_VMD_IS_SEQ(seq)
```

Description

The macro checks that the parameter is a pplib seq. It returns 1 if it is a seq, else if returns 0.

The macro works through variadic macro support.

seq = a possible pplib seq.

returns = 1 if it a seq, else returns 0.

Macro BOOST_VMD_IS_SEQ_D

BOOST_VMD_IS_SEQ_D

Synopsis

```
// In header: <boost/vmd/seq.hpp>

BOOST_VMD_IS_SEQ_D(d, seq)
```

Macro BOOST_VMD_ASSERT_IS_SEQ

BOOST_VMD_ASSERT_IS_SEQ — Asserts that the parameter is a Boost pplib seq.

Synopsis

```
// In header: <boost/vmd/seq.hpp>

BOOST_VMD_ASSERT_IS_SEQ(seq)
```

Description

The macro checks that the parameter is a pplib seq. If it is not a pplib seq, it forces a compiler error.

The macro works through variadic macro support.

The macro normally checks for a pplib seq only in debug mode. However an end-user can force the macro to check or not check by defining the macro BOOST_VMD_ASSERT_DATA to 1 or 0 respectively.

seq = a possible pplib seq.

returns = Normally the macro returns nothing. If the parameter is a pplib seq, nothing is output. For VC++, because there is no sure way of forcing a compiler error from within a macro without producing output, if the parameter is not a pplib seq the macro forces

a compiler error by outputting invalid C++. For all other compilers a compiler error is forced without producing output if the parameter is not a pplib seq.

Macro BOOST_VMD_ASSERT_IS_SEQ_D

BOOST_VMD_ASSERT_IS_SEQ_D

Synopsis

```
// In header: <boost/vmd/seq.hpp>

BOOST_VMD_ASSERT_IS_SEQ_D(d, seq)
```

Header <boost/vmd/tuple.hpp>

```
BOOST_VMD_TUPLE(param)
BOOST_VMD_BEGIN_TUPLE(param)
BOOST_VMD_AFTER_TUPLE(parameter)
BOOST_VMD_IS_TUPLE(param)
BOOST_VMD_ASSERT_IS_TUPLE(tuple)
```

Macro BOOST_VMD_TUPLE

BOOST_VMD_TUPLE — Expands to a tuple of the beginning tuple, and the preprocessor tokens after the beginning tuple, of a macro parameter.

Synopsis

```
// In header: <boost/vmd/tuple.hpp>

BOOST_VMD_TUPLE(param)
```

Description

param = a macro parameter.

returns = the result is a tuple of two elements. If the param does not start with a tuple, both elements of the tuple are empty. If the param does start with a tuple, the first element is the beginning tuple and the second element is the preprocessor tokens after the beginning tuple.

Macro BOOST_VMD_BEGIN_TUPLE

BOOST_VMD_BEGIN_TUPLE — Expands to the beginning tuple of a macro parameter.

Synopsis

```
// In header: <boost/vmd/tuple.hpp>

BOOST_VMD_BEGIN_TUPLE(param)
```

Description

param = a macro parameter.

returns = the preprocessor tokens forming the beginning tuple. If the param does not start with a tuple, the return value is empty.

Macro BOOST_VMD_AFTER_TUPLE

BOOST_VMD_AFTER_TUPLE — Expands to the preprocessor tokens after the beginning tuple of a macro parameter.

Synopsis

```
// In header: <boost/vmd/tuple.hpp>

BOOST_VMD_AFTER_TUPLE(parameter)
```

Description

parameter = a macro parameter.

returns = expands to the preprocessor tokens after the tuple. If the tuple is not found, expands to nothing.

Macro BOOST_VMD_IS_TUPLE

BOOST_VMD_IS_TUPLE — Tests whether a parameter is a Boost PP tuple.

Synopsis

```
// In header: <boost/vmd/tuple.hpp>

BOOST_VMD_IS_TUPLE(param)
```

Description

The macro checks to see if a parameter is a Boost PP tuple. A Boost PP tuple is a parameter enclosed by a set of parenthesis with no prerocessing tokens before or after the parenthesis.

param = a preprocessor parameter

returns = 1 if the param is a Boost PP tuple. 0 if it is not.

Macro BOOST_VMD_ASSERT_IS_TUPLE

BOOST_VMD_ASSERT_IS_TUPLE — Asserts that the parameter is a pplib tuple.

Synopsis

```
// In header: <boost/vmd/tuple.hpp>

BOOST_VMD_ASSERT_IS_TUPLE(tuple)
```

Description

The macro checks that the parameter is a pplib tuple. If it is not a pplib tuple, it forces a compiler error.

The macro works through variadic macro support.

The macro normally checks for a pplib tuple only in debug mode. However an end-user can force the macro to check or not check by defining the macro BOOST_VMD_ASSERT_DATA to 1 or 0 respectively.

tuple = a possible Boost pplib tuple.

returns = Normally the macro returns nothing. If the parameter is a pplib tuple, nothing is output. For VC++, because there is no sure way of forcing a compiler error from within a macro without producing output, if the parameter is not a pplib tuple the macro forces a compiler error by outputting invalid C++. For all other compilers a compiler error is forced without producing output if the parameter is not a pplib tuple.

Design

The initial impetus for creating this library was entirely practical. I had been working on another library of macro functionality, which used Boost PP functionality, and I realized that if I could use variadic macros with my other library, the end-user usability for that library would be easier. Therefore the main design goal of this library is to interoperate variadic macro data with Boost PP in the easiest and clearest way possible.

This led to the original versions of the library as an impetus for adding variadic macro data suppport to Boost PP. While this was being done, but the variadic macro data support had not yet been finalized in Boost PP, I still maintained the library in two modes, either its own variadic data functionality or deferring to the implementation of variadic macros in the Boost PP library.

Once support for variadic data had been added to Boost PP I stripped down the functionality of this library to only include variadic macro support for functionality which was an adjunct to the support in Boost PP. This functionality might be seen as experimental, since it largely relied on a macro which tested for empty input which Paul Mensonides, the author of Boost PP, had published on the Internet, and which by the very nature of the C++ preprocessor is slightly flawed but which was the closest approximation of such functionality which I believed could be made. I had to tweak this macro somewhat for the Visual C++ preprocessor, whose conformance to the C++ standard for macro processing is notably incorrect in a number of areas. But I still felt this functionality could be used in select situations and might be useful to others. Using this functionality I was able to build up some other macros which tested for the various Boost PP data types. I also was able to add in functionality, based on Paul Mendsonides excellent work, for handling tuples in preproccessing data.

All of this particular functionality is impossible to do effectively without the use of variadic macros. But I had kept these features at a minimum because of the difficulty of using variadic macros with compilers, most notably Visual C++, whose implementation of variadic macros is substandard and therefore very difficult to get to work correctly when variadic macros must be used.

I then realized that if I am going to have a library which takes advantage of variadic macros I should see what I could do in the area of parsing preprocessor data. This has led to a reorganization of the library as a set of macros largely for parsing preprocessor data. All of this is now built on top of my use of the almost perfect checking for emptiness which Paul Mensonides originally created.

Compilers

I have tested this library using gcc/MingW, VC++, and clang targeting gcc on Windows. The compilers tested are gcc 3.3.3, 3.4.2, 3.4.5, 4.3.0, 4.4.0, 4.5.0-1, 4.5.2-1, 4.6.0, 4.6.1, 4.6.2, 4.7.0, 4.7.2, 4.8.1, VC++ 8.0, 9.0, 10.0, 11.0, 12.0, and the latest clang build. Other compilers which currently should work with this library are gcc on Linux and clang.

For VC++ 8.0 the BOOST_VMD_IS_BEGIN_TUPLE and BOOST_VMD_IS_EMPTY macros take a single parameter rather than variadic data because this version of VC++ does not accept variadic data which may be empty.

The compilers supported are those which are deemed to offer C99/C++11 variadic macro support for Boost PP as represented by the BOOST_PP_VARIADICS macro.

History

Version 1.7

- The library has been reengineered to provide vastly added functionality. This includes:
  - Adding functionality for parsing v-types.
  - Adding functionality for parsing sequencies of v-types.
  - Adding improved ASSERT macros.
  - Adding BOOST_VMD_EMPTY and BOOST_VMD_IDENTITY.

Version 1.6

- Stripped off all functionality duplicated by the variadic macro functionality added to Boost PP.
- Removed the notion of 'native' and 'pplib' modes.
- Use the BOOST_PP_VARIADICS macro from the Boost PP library to determine variadic macro availability and removed the native macro for determining this from this library.
- Updated documentation, especially to give fuller information of the use of the BOOST_VMD_EMPTY macro and its flaw and use with Visual C++.
- Changed the directory structure to adhere to the Modular Boost structure.

Version 1.5

- Added macros for verifying Boost PP data types.
- Added macros for detecting and removing beginning parens.
- Added a macro for testing for the emptiness of a parameter.
- Added support for individual header files.
- Added support for 'native' and 'pplib' modes.
- Added control macros for controlling the variadic macro availability, mode, and data verification.

Version 1.4

- Removed internal dependency on BOOST_PP_CAT and BOOST_PP_ADD when using VC++.

Version 1.3

- Moved version information and history into the documentation.
- Separate files for build.txt in the doc sub-directory and readme.txt in the top-level directory.
- Breaking changes
  - The name of the main header file is shortened to 'vmd.hpp'.
  - The library follows the Boost conventions.
    - Changed the filenames to lower case and underscores.

- The macros now start with BOOST_VMD_ rather than just VMD_ as previously.

Version 1.2

- Added a readme.txt file.
- Updated all jamfiles so that the library may be tested and docs generated from its own local directory.

Version 1.1

- Added better documentation for using variadic data with Boost PP and VMD.

Version 1.0

Initial version of the libary.

Acknowledgements

First and foremost I would like to thank Paul Mensonides for providing advice, explanation and code for working with variadic macros and macros in general. Secondly I would like to thank Steve Watanabe for his help, code, and explanations. Finally I have to acknowledge that this library is an amalgam of already known techniques for dealing with variadic macros themselves, among which are techniques published online by Paul Mensonides. I have added design and some cleverness in creating the library but I could not have done it without the previous knowledge of others.

Index

Macro BOOST_VMD_AFTER_SEQ_D
  BOOST_VMD_AFTER_SEQ_D
Macro BOOST_VMD_AFTER_TUPLE
  BOOST_VMD_AFTER_TUPLE
Macro BOOST_VMD_ARRAY
  BOOST_VMD_ARRAY
Macro BOOST_VMD_ASSERT
  BOOST_VMD_ASSERT
Macro BOOST_VMD_ASSERT_IS_ARRAY
  BOOST_VMD_ASSERT_IS_ARRAY
Macro BOOST_VMD_ASSERT_IS_EMPTY
  BOOST_VMD_ASSERT_IS_EMPTY
Macro BOOST_VMD_ASSERT_IS_IDENTIFIER
  BOOST_VMD_ASSERT_IS_IDENTIFIER
Macro BOOST_VMD_ASSERT_IS_IDENTIFIER_D
  BOOST_VMD_ASSERT_IS_IDENTIFIER_D
Macro BOOST_VMD_ASSERT_IS_LIST
  BOOST_VMD_ASSERT_IS_LIST
Macro BOOST_VMD_ASSERT_IS_LIST_D
  BOOST_VMD_ASSERT_IS_LIST_D
Macro BOOST_VMD_ASSERT_IS_NUMBER
  BOOST_VMD_ASSERT_IS_NUMBER
Macro BOOST_VMD_ASSERT_IS_SEQ
  BOOST_VMD_ASSERT_IS_SEQ
Macro BOOST_VMD_ASSERT_IS_SEQ_D
  BOOST_VMD_ASSERT_IS_SEQ_D
Macro BOOST_VMD_ASSERT_IS_TUPLE
  BOOST_VMD_ASSERT_IS_TUPLE
Macro BOOST_VMD_BEGIN_ARRAY
  BOOST_VMD_BEGIN_ARRAY
Macro BOOST_VMD_BEGIN_IDENTIFIER
  BOOST_VMD_BEGIN_IDENTIFIER
  BOOST_VMD_MAP_
Macro BOOST_VMD_BEGIN_IDENTIFIER_D
  BOOST_VMD_BEGIN_IDENTIFIER_D
Macro BOOST_VMD_BEGIN_LIST
  BOOST_VMD_BEGIN_LIST
Macro BOOST_VMD_BEGIN_LIST_D
  BOOST_VMD_BEGIN_LIST_D
Macro BOOST_VMD_BEGIN_NUMBER
  BOOST_VMD_BEGIN_NUMBER
Macro BOOST_VMD_BEGIN_SEQ
  BOOST_VMD_BEGIN_SEQ
Macro BOOST_VMD_BEGIN_SEQ_D
  BOOST_VMD_BEGIN_SEQ_D
Macro BOOST_VMD_BEGIN_TUPLE
  BOOST_VMD_BEGIN_TUPLE
Macro BOOST_VMD_EMPTY
  BOOST_VMD_EMPTY
Macro BOOST_VMD_IDENTIFIER
  BOOST_VMD_IDENTIFIER
  BOOST_VMD_MAP_
Macro BOOST_VMD_IDENTIFIER_D
  BOOST_VMD_IDENTIFIER_D
Macro BOOST_VMD_IDENTITY
  BOOST_VMD_IDENTITY
Macro BOOST_VMD_IDENTITY_RESULT
  BOOST_VMD_IDENTITY_RESULT

Macro BOOST_VMD_IS_ARRAY
  BOOST_VMD_IS_ARRAY
Macro BOOST_VMD_IS_BEGIN_ARRAY
  BOOST_VMD_IS_BEGIN_ARRAY
Macro BOOST_VMD_IS_BEGIN_IDENTIFIER
  BOOST_VMD_IS_BEGIN_IDENTIFIER
  BOOST_VMD_MAP_
Macro BOOST_VMD_IS_BEGIN_IDENTIFIER_D
  BOOST_VMD_IS_BEGIN_IDENTIFIER_D
Macro BOOST_VMD_IS_BEGIN_LIST
  BOOST_VMD_IS_BEGIN_LIST
Macro BOOST_VMD_IS_BEGIN_LIST_D
  BOOST_VMD_IS_BEGIN_LIST_D
Macro BOOST_VMD_IS_BEGIN_NUMBER
  BOOST_VMD_IS_BEGIN_NUMBER
Macro BOOST_VMD_IS_BEGIN_SEQ
  BOOST_VMD_IS_BEGIN_SEQ
Macro BOOST_VMD_IS_BEGIN_SEQ_D
  BOOST_VMD_IS_BEGIN_SEQ_D
Macro BOOST_VMD_IS_BEGIN_TUPLE
  BOOST_VMD_IS_BEGIN_TUPLE
Macro BOOST_VMD_IS_EMPTY
  BOOST_VMD_IS_EMPTY
Macro BOOST_VMD_IS_EMPTY_ARRAY
  BOOST_VMD_IS_EMPTY_ARRAY
Macro BOOST_VMD_IS_EMPTY_ARRAY_IRESULT
  BOOST_VMD_IS_EMPTY_ARRAY_IRESULT
Macro BOOST_VMD_IS_EMPTY_LIST
  BOOST_VMD_IS_EMPTY_LIST
Macro BOOST_VMD_IS_EMPTY_LIST_D
  BOOST_VMD_IS_EMPTY_LIST_D
Macro BOOST_VMD_IS_EMPTY_LIST_D_IRESULT
  BOOST_VMD_IS_EMPTY_LIST_D_IRESULT
Macro BOOST_VMD_IS_EMPTY_LIST_IRESULT
  BOOST_VMD_IS_EMPTY_LIST_IRESULT
Macro BOOST_VMD_IS_IDENTIFIER
  BOOST_VMD_IS_IDENTIFIER
  BOOST_VMD_MAP_
  BOOST_VMD_MAP_PLANEGEOMETRY_1_SQUARE
  BOOST_VMD_MAP_PLANEGEOMETRY_2_TRIANGLE
  BOOST_VMD_MAP_PLANEGEOMETRY_3_CIRCLE
Macro BOOST_VMD_IS_IDENTIFIER_D
  BOOST_VMD_IS_IDENTIFIER_D
Macro BOOST_VMD_IS_LIST
  BOOST_VMD_IS_LIST
Macro BOOST_VMD_IS_LIST_D
  BOOST_VMD_IS_LIST_D
Macro BOOST_VMD_IS_NUMBER
  BOOST_VMD_IS_NUMBER
Macro BOOST_VMD_IS_PARENS_EMPTY
  BOOST_VMD_IS_PARENS_EMPTY
Macro BOOST_VMD_IS_SEQ
  BOOST_VMD_IS_SEQ
Macro BOOST_VMD_IS_SEQ_D
  BOOST_VMD_IS_SEQ_D
Macro BOOST_VMD_IS_TUPLE
  BOOST_VMD_IS_TUPLE
Macro BOOST_VMD_LIST