The Variadic Macro Data Library 1.6

Edward Diener

Table of Contents

Introduction

Welcome to version 1.6 of the Variadic Macro Data library.

The Variadic Macro Data library, or VMD for short, is a library of variadic macros which enhance the functionality in the Boost preprocessor library (Boost PP). The library may also be seen as a repository for future enhancements to the Boost PP library functionality using variadic macros.

The functionality of the library may be summed up as:

1. Provide a better way of testing for empty parameters.

2. Provide ways for testing for parens and for removing parens.

3. Provide support for verifying Boost PP data types.

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.

The library is dependent on Boost PP.

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.

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 that compiler support for variadic macros exist. This macro can be set by a user of the Boost PP library, as well as 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 BOOST_PP_VARIADICS macro. The end-user of VMD can also manually use the macro BOOST_PP_VARIADICS to turn on or off compiler support for variadic macro data in the VMD library. When BOOST_PP_VARIADICS is set to 0 variadic macro data is not supported in the VMD library, otherwise when BOOST_PP_VARIADICS is set to non-zero it is supported in the VMD library. This same macro can be used to determine if VMD supports variadic macros for a particular compiler.

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.

Functionality

The design of Boost PP allows data, in the form of preprocessor tokens, to be grouped together into various data types, any one of which can be treated as a single preprocessor argument to a macro. A number of Boost PP macros accept data as a single argument in the form of one of these data types. Each of these data types also has its own rich set of macros to manipulate the data.

VMD offers macro functionality, using variadic macros, centered on the verification of Boost PP data types, on the detection and removal of beginning parentheses in a macro parameter, and on the determination of whether a parameter is empty or not.

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 sequence data type.

I have used names in order to mimic the naming of Boost PP as closely as possible.

Functional groups

The macros in VMD can best be explained as falling into three groups. These are:

1. Macros which add low-level preprocessor functionality.
2. Macros which verify a Boost PP data type.
3. Macros which control internal variadic data functionality.

A further general explanation of each of these groups follow, while a specific explanation for each macro can be found in the reference section.

Low-level preprocessor functionality

There are macros which add low-level preprocessor functionality related to Boost PP through the use of variadic macros. These can be divided into two groups:

- BOOST_VMD_EMPTY = Testing for empty input
- BOOST_VMD_IS_BEGIN_PARENS and BOOST_VMD_REMOVE_PARENS = Handling parenthensis

Testing for empty input

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 some limited inputs or emptiness. 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. 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 slightly different code to work with the Visual C++ compiler since Paul's code would not work as is because of deficiencies in the Visual C++ preprocessor.

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

The one situation where the macro may not work properly is if its input is a function-like macro name. The problem that will occur for a standard C++ conforming preprocessor is that if the function-like macro takes two or more parameters, a compiler error will occur. Otherwise the BOOST_VMD_IS_EMPTY macro will work properly even if the input is a function-like macro name with 0 or 1 parameter.

Macro Flaw with Visual C++

The VC++ preprocessor is not a standard C++ conforming preprocessor in at least two situations which cause the BOOST_VMD_IS_EMPTY macro to not work properly using VC++ when the input is 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.

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

FMACRO(1)
```

should give a compiler error but when invoked using VC++ it only gives a warning and VC++ continues macro substitution with 'y' as a placemarker preprocessing token.

The second situation 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 exmaple:

```
#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:

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

```
FMACRO3 FMACRO2()
```

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

What these preprocessor flaws in the VC++ compiler mean is that if a function-like macro name is passed as input to the macro, and the function-like macro when expanded is a set of parentheses '()' with 0 or more tokens inside them, BOOST_VMD_IS_EMPTY will erroneously return 1 rather than 0 when using the Visual C++ compiler.

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 considered the macro worthwhile to use with the vast majority of possible input. Obviously the macro should be used by the end-user when the possible input to it is constrained but it can still serve a purpose in preprocessor programming. The macro is used extensively in the assert-like macros explained further on in the documentation and macro programmers may find this macro useful in their own programming efforts despite the slight flaw in the way that it works.

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

Testing and removing parens

A common need when using macros and Boost PP is determining whether a parameter begins with a set of parenthesis, within which tokens may exist. This is the form of a Boost PP tuple. All Boost PP data types begin this way, and both Boost PP arrays and lists are tuples of a certain form, while a Boost PP seq is a series of one or more single element tuples. Once again I have taken the implementation from code which Paul Mensonides posted and have changed it slightly for Visual C++.

The macro `BOOST_VMD_IS_BEGIN_PARENS` takes as its input variadic data and determines whether it begins with a set of parenthesis. It returns 1 if its input begins with a set of parenthesis, else it returns 0. The macro is a variadic macro which make take any input[2].

There may be other tokens after the set of parameters that begins the input, and the macro still returns 1.

Another common need when using macros and Boost PP is to remove the set of parenthesis from the beginning of a parameter. This is what `BOOST_VMD_REMOVE_PARENS` does. if the parameter does not contain a set of beginning parenthesis, the input parameter is returned as is, else the input parameter is returned with the beginning parenthesis removed. No other parenthesis after the possible beginning parenthesis is removed.

The end-user of VMD can include the individual header file 'vmd_is_begin_parens.hpp' instead of the general header file 'vmd.hpp' for using the BOOST_VMD_IS_BEGIN_PARENS macro and the individual header file 'vmd_remove_parens.hpp' instead of the general header file 'vmd.hpp' for using the BOOST_VMD_REMOVE_PARENS macro.

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

Verifying Boost PP data types

There is no way to determine whether a macro parameter is a given Boost PP data type. But one can come close to this functionality using variadic macros and the test for emptiness in the VMD library which the BOOST_VMD_IS_EMPTY macro gives. Therefore I have developed four macros, for each of the four Boost PP data types, which will produce a compiler error if a parameter is not a given data type, else will output nothing. These macros are:

1. `BOOST_VMD_ASSERT_IS_ARRAY`(param), asserts that 'param' is a Boost PP array

2. `BOOST_VMD_ASSERT_IS_LIST`(param), asserts that 'param' is a Boost PP list

3. `BOOST_VMD_ASSERT_IS_SEQ`(param), asserts that 'param' is a Boost PP seq

4. `BOOST_VMD_ASSERT_IS_TUPLE`(param), asserts that 'param' is a Boost PP tuple

Each of these act like a C++ assert-like macro. They output nothing if the 'param' is of the known type, else produce a compiler error. Furthermore these macros, like 'assert-like' macros, only check their 'param' in debug mode. This, however, can be overridden by the BOOST_VMD_ASSERT_DATA macro. If it is defined and set to 1, the macros will always check their 'param', otherwise if it is set to 0 the 'param' is never checked.

The assert macros can be used in code to test whether a 'param' is of the expected Boost PP data type. It is not possible to create equivalent macros which return 1 when the 'param' is of the correct type, else return 0, because of the assert macros' necessary reliance on BOOST_VMD_IS_EMPTY, which even in the best of conditions is slightly flawed. However the flaw in BOOST_VMD_IS_EMPTY, where a function-like macro name could produce a compiler error, is made use of in the assert macros so that the compiler error produced in that case will only occur when the assert macro verifies that its 'param' is not the Boost PP data type being checked. Still there is the very slight flaw in the assert macros where its reliance on BOOST_VMD_IS_EMPTY could run into input being checked where a function-like macro name will produce the incorrect result from the BOOST_VMD_IS_EMPTY macro when using VC++. However this flaw is thought to be so slight, and an end-user's chance of inputting such data so minimal, that it was felt that such assert-like macros would be beneficial to macro programmers using Boost PP and VMD.

Another point to be made is that Boost PP array and list are also tuples. So passing an array or list to BOOST_VMD_ASSERT_IS_TUPLE will assert that either one of those Boost PP data types is a valid tuple by not producing a compiler error.

A final point concerns using the assert macros with Visual C++. There is no way which I have been able to discover to produce a compiler error from within a macro using Visual C++ without producing invalid C++ output to trigger the error. If the VC++ compiler subsequently sees this output after the preprocessing stage is complete it will create an error message.

Because of this the promise to produce no output in case of error from the assert macros must be amended for Visual C++. The compiler error when using Visual C++, and an assert macro fails, could be masked by using one of the assert macros in a situation where no output is generated by an enclosing macro. This case would occur if the result of one of the assert macros were embedded in another macro, and the embedded macro just ignores any "output" from VC++ when the assert macro fails. An admittedly artifical example using VC++ might be:

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

#define IGNORE_TUPLE_CHECK(vcoutput)
#define USE_TUPLE(atuple) IGNORE_TUPLE_CHECK(BOOST_VMD_ASSERT_TUPLE(atuple)) PROCESS_TUPLE(atuple)
```

When USE_TUPLE is invoked using VC++, if the input is not actually a Boost PP tuple BOOST_VMD_ASSERT_TUPLE will output incorrect C++. But since that incorrect output is passed to another macro which just ignores it, the output will never appear after the preprocessing stage so no compiler error will occur.

Individual header files can be used for each of the assert macros instead of including the general header file 'vmd.hpp'. The individual header files are:

- 'vmd_assert_is_array.hpp' for the BOOST_VMD_ASSERT_IS_ARRAY macro

- 'vmd_assert_is_list.hpp' for the BOOST_VMD_ASSERT_IS_LIST macro
- 'vmd_assert_is_seq.hpp' for the BOOST_VMD_ASSERT_IS_SEQ macro
- 'vmd_assert_is_tuple.hpp' for the BOOST_VMD_ASSERT_IS_TUPLE macro.

Controlling internal usage

VMD has a few object-like macros which the end-user of the library can use to determine or change the way variadic macros are used in the library.

The macro BOOST_PP_VARIADICS is part of the Boost PP library, not part of VMD. It is used to denote whether variadic data support exists for the particular compiler the end-user is using. VMD also uses this macro to determine whether variadic data support exists. An end-user of VMD can use this macro in his own design to determine whether or not variadic macros are supported. Furthermore an end-user of VMD can set this macro to 0 or non-zero, before including a VMD header file, to force VMD to treat the particular compiler being used as not supporting or supporting variadic macros.

The macro BOOST_VMD_ASSERT_DATA controls whether or not an assert macro will check its data. The default is that in compiler debug mode it will check the data while in compiler release mode it will not check its data. The end-user can change this by setting the macro to 0 to not check the data, or non-zero to check the data before including a VMD header file, or check the value if necessary after including a VMD header file.

Variadic Macro Data Reference

Header <boost/vmd/vmd_assert_is_array.hpp>

```
BOOST_VMD_ASSERT_IS_ARRAY(array)
```

Macro BOOST_VMD_ASSERT_IS_ARRAY

BOOST_VMD_ASSERT_IS_ARRAY — Asserts that the parameter is a pplib array.

Synopsis

```
// In header: <boost/vmd/vmd_assert_is_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.

There is no completely fool-proof way to check if a parameter is empty without possible producing a compiler error if it is not. Because a macro checking if a parameter is a pplib array needs to perform such a check, the best that one can do is to create a compiler error if a parameter is not a pplib array rather than having a macro which returns 1 or 0, depending on whether a parameter is a pplib array.

Header <boost/vmd/vmd_assert_is_list.hpp>

```
BOOST_VMD_ASSERT_IS_LIST(list)
```

Macro BOOST_VMD_ASSERT_IS_LIST

BOOST_VMD_ASSERT_IS_LIST — Asserts that the parameter is a pplib list.

Synopsis

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

BOOST_VMD_ASSERT_IS_LIST(list)
```

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.

list = 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.

There is no completely fool-proof way to check if a parameter is empty without possible producing a compiler error if it is not. Because a macro checking if a parameter is a pplib list needs to perform such a check, the best that one can do is to create a compiler error if a parameter is not a pplib list rather than having a macro which returns 1 or 0, depending on whether a parameter is a pplib list.

Header <boost/vmd/vmd_assert_is_seq.hpp>

```

BOOST_VMD_ASSERT_IS_SEQ(seq)
```

Macro BOOST_VMD_ASSERT_IS_SEQ

BOOST_VMD_ASSERT_IS_SEQ — Asserts that the parameter is a pplib seq.

Synopsis

```
// In header: <boost/vmd/vmd_assert_is_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.

There is no completely fool-proof way to check if a parameter is empty without possible producing a compiler error if it is not. Because a macro checking if a parameter is a pplib seq needs to perform such a check, the best that one can do is to create a compiler error if a parameter is not a pplib seq rather than having a macro which returns 1 or 0, depending on whether a parameter is a pplib seq.

Header <boost/vmd/vmd_assert_is_tuple.hpp>

```
BOOST_VMD_ASSERT_IS_TUPLE(tuple)
```

Macro BOOST_VMD_ASSERT_IS_TUPLE

BOOST_VMD_ASSERT_IS_TUPLE — Asserts that the parameter is a pplib tuple.

Synopsis

```
// In header: <boost/vmd/vmd_assert_is_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 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.

There is no completely fool-proof way to check if a parameter is empty without possible producing a compiler error if it is not. Because a macro checking if a parameter is a pplib tuple needs to perform such a check, the best that one can do is to create a compiler error if a parameter is not a pplib tuple rather than having a macro which returns 1 or 0, depending on whether a parameter is a pplib tuple.

Header <boost/vmd/vmd_is_begin_parens.hpp>

```
BOOST_VMD_IS_BEGIN_PARENS(...)
```

Macro BOOST_VMD_IS_BEGIN_PARENS

BOOST_VMD_IS_BEGIN_PARENS — Tests whether a parameter begins with a set of parentheses.

Synopsis

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

BOOST_VMD_IS_BEGIN_PARENS(...)
```

Description

The macro checks to see if the parameter begins with a set of parentheses surrounding any tokens.

.... = variadic param(s)

returns = 1 if the param begins with a set of parentheses, 0 if it does not.

The macro works through variadic macro support.

The code for the non-VC++ version of this is taken from a posting by Paul Mensonides.

This macro is not a test for a parameter which is only a single set of parentheses surrounding any tokens (a Boost pplib tuple) since the parameter may have other tokens following the beginning set of parentheses and it will still return 1.

There is no completely safe way to test whether the param is a tuple. At best one can use BOOST_VMD_ASSERT_IS_TUPLE to cause a compiler error if the parameter is not a tuple.

Header <boost/vmd/vmd_is_empty.hpp>

```


BOOST_VMD_IS_EMPTY(...)
```

Macro BOOST_VMD_IS_EMPTY

BOOST_VMD_IS_EMPTY — Tests whether its input is empty or not.

Synopsis

```
// In header: <boost/vmd/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.

Header <boost/vmd/vmd_remove_parens.hpp>

```
BOOST_VMD_REMOVE_PARENS(param)
```

Macro BOOST_VMD_REMOVE_PARENS

BOOST_VMD_REMOVE_PARENS — Removes the set of parens from the start of a parameter if it has any.

Synopsis

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

BOOST_VMD_REMOVE_PARENS(param)
```

Description

param = a macro parameter.

returns = the parameter with the beginning set of parens removed. If the parameter has no beginning set of parameters, the parameter is returned as is. If there are further sets of parens after the beginning set of parameters, they are not removed.

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 functionlity, based on Paul Mendsonides excellent work, for handling parens in preproccessing data.

All of this particular functionality is impossible to do effectively without the use of variadic macros. But I have 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.

Nonetheless I would like to keep this library as an adjunct to the functionlity in Boost PP for using variadic macros. I believe that others may jhave implemented their own expanded functionality using variadic macros which may further be added to the library in the future.

Compilers

I have tested this library using gcc/MingW and VC++ 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 and VC++ 8.0, 9.0, 10.0, 11.0. Other compilers which currently should work with this library are gcc on Linux and clang. Other compilers which may work with this library are Digital Mars C++, and Borland C++ Builder 6/Codegear C++.

For VC++ 8.0 the BOOST_VMD_IS_BEGIN_PARENS 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 variadic macro support for Boost PP as represented by the BOOST_PP_VARIADICS macro.

History

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